FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

A SysML-based Design Flow for Digital VLSI Circuits

Manuel José Santos Oliveira

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Supervisor: José Carlos Alves

Second Supervisor: António Pacheco

Third Supervisor: Miguel Falcão

July 24, 2015

U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

MIEEC - MESTRADO INTEGRADO EM ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES | **2014/2015**

A Dissertação intitulada

"A SysML-based Design Flow for Digital VLSI Circuits"

foi aprovada em provas realizadas em 17-07-2015

o júri

Presidente Professor Doutor José Alfredo Ribeiro da Silva Matos
Professor Catedrático do Departamento de Engenharia Eletrotécnica e de Computadores da Faculdade de Engenharia da Universidade do Porto

Professor Doutor Mário Pereira Véstias
Professor Coordenador do Departamento de Engenharia Eletrónica e Telecomunicações e de Computadores do Instituto Superior de Engenharia de Lisboa

Professor Doutor José Carlos dos Santos Alves
Professor Associado do Departamento de Engenharia Eletrotécnica e de Computadores da Faculdade de Engenharia da Universidade do Porto

O autor declara que a presente dissertação (ou relatório de projeto) é da sua exclusiva autoria e foi escrita sem qualquer apoio externo não explicitamente autorizado. Os resultados, ideias, parágrafos, ou outros extratos tomados de ou inspirados em trabalhos de outros autores, e demais referências bibliográficas usadas, são corretamente citados.

Autor - Manuel José Santos Oliveira

Faculdade de Engenharia da Universidade do Porto

© Manuel José Santos Oliveira, 2015

Resumo

Em empresas competitivas como a Synopsys, permitir que janelas de oportunidade se fechem traduz-se em perdas económicas bastante elevadas. Esta perda é tipicamente provocados por atrasos no desenvolvimento dos produtos. No sentido de minimizar atrasos, por vezes a documentação da micro-arquitetura (documentação interna do produto) é colocada em segundo plano. Esta decisão leva a que a única maneira de se fazer a manutenção de código, seja para implementação de novas funcionalidades ou alteração das já implementadas, seja através da leitura do código, dificultando este tipo de operações, recorrentes no desenvolvimento de IPs.

O SysML é uma linguagem de modelação baseada em UML criada para sistemas de engenharia que disponibiliza um conjunto de diagramas que ajudam a documentar o projeto de sistemas VLSI ao mesmo tempo que ajuda a melhorar a comunicação interna das equipas. No entanto, apesar de melhorar a qualidade da documentação, não reduz o tempo de desenvolvimento dos projetos.

Este trabalho tem como objetivo utilizar um subconjunto de diagramas do SysML para gerar código de descrição de hardware reconfigurável e sintetizável aproveitando, desta forma, as capacidades do SysML para documentar SoC, ao mesmo tempo que se interliga o código com a documentação. Esta abordagem pretende motivar as equipas a investirem mais tempo na qualidade da documentação produzida ao mesmo tempo que diminui o tempo de desenvolvimento, uma vez que, o código resultaria diretamente do modelo SysML.

Este trabalho foi sugerido pela Synopsys tendo por base o fluxo de projeto típico da empresa e com especial atenção por aspetos próprios da empresa no desenvolvimento dos seus produtos.

O resultado do trabalho realizado consiste numa ferramenta de geração automática de código Verilog 2001, tendo por base modelos em SysML criados na ferramenta comercial da Sparx - Enterprise Architect.

Abstract

Missing opportunity windows in competitive companies such as Synopsys can cause substantial economic losses. This is typically consequence of a delay in the production process. In order to minimize that postponement, the micro-arquitecture's documentation (internal documentation of the product) may be pushed to the background. Doing so implies that the only way to do code maintenance is by fully reading it. In spite of this, typical operations in the development of IPs, such as the implementation of new functionalities or changing an already existing function, become harder to implement.

SysML is a modulation language based on UML created to engineering system applications. This provides a set of diagrams that can be useful in documenting a project of VLSI systems and also helps to improve the communication between internal teams. Nevertheless, although it increases the documentation quality, it does not reduce the project development time.

The main objective of this word is to use a subset of SysML diagrams to generate the description code of the resettable and synthesizable hardware. This way, it is possible to take advantage of the SysML capabilities to document SoC, while the code and the documentation are connecting. This approach is intended as a motivation to the teams for investing more time in improving the documentation quality, and this leads to reducing the development time, because the code would result directly from SysML model.

This essay was proposed by Synopsys, by reference the company design flow and with special attention to its own aspects in the development of products.

The result of the accomplished work consist in a tool for automatic generation of Verilog 2001 code, based on SysML models generated by the commercial tool of Sparx - Enterprise Architect.

Agradecimentos

Aos meus orientadores, o Prof. José Carlos Alves, o Eng.º António Pacheco e ao Eng.º Miguel Falcão, o meu muito obrigado por todo o apoio, disponibilidade e empenho prestado ao longo deste trabalho.

A minha família por todo o apoio prestado.

Aos meus companheiros de jornada académica, pessoas que, em alguma altura da minha vida académica, me apoiaram e ajudaram a concluir esta grande e importante etapa. Em especial a companheiros e amigos que estiveram sempre dispostos a ouvir as minhas lamentações e preocupações: José Perdiz, Sofia Inácio, José Araújo, Luís Amorim e Isabel Fragoso. E a tantos outros que sempre me apoiaram e ajudaram: Bruno Augusto, Ana Miranda, André Couto, Daniel Jorge, Carlos Nunes, Miguel Fernandes, Daniela Carmo, Lúcia Vaz, Ciro Monteiro, António Silva e Carlos Rodrigues.

Não posso deixar de realçar três grandes amigos desta vida académica; amigos de sempre e para sempre. Amigos com quem partilhei momentos que nunca esquecerei: Inês Teixeira, Eduardo Fernandes e Ricardo Calçarão. Levar-vos-ei sempre "comigo pr'a vida".

O meu agradecimento especial à Inês Teixeira e ao Miguel Fernandes por terem despendido o seu precioso tempo para reverem este trabalho, demonstrando sem dúvida a seu grande apreço por mim.

Porque a vida não é só faculdade, o meu muito obrigado aos meus grandes amigos do grupo Na Onda da Fé e do coro de Santa Cecília pelos momentos de partilha e descontracção depois de semanas difíceis de trabalho, pelo apoio, carinho e amizade que sempre tiveram para comigo: Mariana Maia, Anacleta Morais, Liliana Oliveira, Hélder Barros, Salomé Pereira, Leandro Nogueira, Carina Pinto, Andreia Patrícia, Mafalda Pereira e Filipa Teixeira.

Aos meus grandes amigos Pedro Tavares e André Tavares. Há amizades que a distância e o tempo não separa.

E por último, mas dos mais importantes, o meu enorme agradecimento àquela que partilha comigo a jornada da vida. Um ombro amigo incondicional, a melhor amiga, a namorada, Vera Teixeira. Obrigado pelo apoio, motivação, inspiração, compreensão, amizade e o amor que tens para comigo desde sempre.

Manuel Oliveira

"Porque eu sou do tamanho do que vejo
E não, do tamanho da minha altura... "

Alberto Caeiro

Contents

List of Figures

List of Tables

Abbreviations

ASIC	Application-Specific Integrated Circuit
CAN	Controller Area Network
CPU	Central Processing Unit
DMA	Direct Memory Access
DSE	Design Space Exploration
EA	Enterprise Architect
EDA	Electronic Design Automation
eDMA	PCI express DMA
FPGA	Field-Programmable Gate Array
FSM	Finite-State Machine
HDL	Hardware Description Language
HLS	High Level Synthesis
HTML	HyperText Markup Language
IBM	International Business Machines
IP	Intellectual Property
MARTE	Modelling and Analysis of Real-Time and Embedded Systems
MDD	Model-driven Development
PCIe	Peripheral Component Interconnect Express
RTES	Real Time Embedded Systems
RTL	Register-transfer-level
SNPS	Synopsys
SoC	System on a Chip
SoPC	System on a Programmable Chip
SysML	Systems Modeling Language
TLP	Transaction Layer Packet
UML	Unified Modeling Language
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits
VLSI	Veri-large -scale integration
XML	Extensible Markup Language

Chapter 1

Introduction

1.1 Context

In the last decades, the complexity of digital circuits has increased significantly and so have their design costs, where a very significant part is allocated to time-consuming verification procedures. To minimize these, it is necessary to develop new methods to improve the time-to-market. Besides the verification issues, the time used to customize a design to meet the customer's requirements and elaborate the documentation is also an important way to improve the time to market. In competitive companies, such as Synopsys, the windows of opportunity are very short. To meet those targets, sometimes, the micro-architecture documentation is not the first priority and because of this lack of description, it is the Register Transfer Level (RTL) code the document that describes the project itself. As a consequence to this fact, the code is harder to read and when needed to be corrected or changes, this process is more time consuming than it could be and error prone.

The development of a digital VLSI design begins with surveys of requirements necessary to meet specifications set by various stakeholders. In most cases, these specifications are unclear and incomplete, which can lead to ambiguous situations that, invariably, increase design time, causing economic losses.

After gathering the list of requirements, it is necessary to organize the project and divide the system into blocks, dictated by functional or technological requirements. Due to carrying out this 'dividing to rule' strategy, it is possible to develop reusable modules, that are essential to increasing productivity in future projects. Both modules and project are parameterized to allow the creation of different profiles, adapting to project stakeholders. All this information is typically documented textually.

Invariably, during implementation, it is necessary to make minor adjustments which result in inconsistencies with the documentation. Usually, these adjustments are not immediately reflected in the technical documentation. This involves risking the extension of the project duration, with potential severe financial consequences. From a business point of view, it is crucial to maintain the documentation up to date but still minimizing unnecessary effort.

To overcome these problems, a graphical language - SysML - can be used to describe a project at different abstraction levels (structural, behavioral). The goal is to clarify and organize ideas maintaining the synchronization between stakeholders and designers. SysML is based on UML, inheriting several concepts used in software design for the specification and design of systems, in a broader sense. This graphic language defines several diagrams that help in the description of problem and solution. The SysML is divided into four main pillars: requirements, behavior, structure and parameterization, with specific diagrams for each one. If it would be possible to use the SysML diagrams to generate automatically the Verilog code from a graphical view of the design, specified in SysML, the development time would be improved.

This work intends to give a contribution for improving the productivity in the digital design flow by allowing the use of SysML to describe a system in an abstract and easy to maintain graphical perspective. The purpose is to develop an EDA tool to automate the writing of HDL synthesizable codes, taking as source the specifications of the system functionality and design requirements in SysML. This will decrease the manual intervention in crucial stages of the design flow and thus contribute to improve the productivity of the design team. This work was proposed by the host company Synopsys, as the continuation of an exploratory development done during a summer job, from July to September 2014. This preliminary work was initially proposed as an effort to normalize the style of technical documentation and maintain it up to date with the system under design. The experience with the Enterprise Architect modeling and design tool and the limited features to generate HDL code from state charts described in SysML, led to further analysis of this high level language, to automate the generation of HDL at different abstraction levels. Not only has this enabled the construction of parameterized structural top-level models, but also it has allowed the synthesis of Finite-State Machines. The present work was developed in the light of an internship in Synopsys Maia facilities, which provided access to the required EDA Synopsys tools. The work was divided in two main components. The first part focuses on the automatic generation of a hierarchical structure of modules, with the parameterization specified by the user requirements. The second phase approaches the automation of the generation of control-dominated systems, to create HDL models of finite state machines from abstract SysML diagrams. These solutions will reduce the development time and minimize the possibility of human errors. The targets defined for this project are:

- Create rules for the Enterprise Architect to model a synthesized and parameterized SoC, in SysML;
- Develop a tool that automatically generates Verilog code using the SysML diagrams, verifying the consistency between model and code;
- Create rules for the Enterprise Architect to model a FSM in SysML;
- Implement a new feature to support automatic generation of FSM.

The AutoGen tool (Automatic Generator of Verilog Code) was developed in Python 2.7.9, because of its simplicity, public domain and portability among different operating systems. The

Enterprise Architect was used to create the SysML model which will be used as an input for the AutoGen tool.

1.2 About Synopsys

1.2.1 The Company

Synopsys is the leader in electronic design automation (EDA) and semiconductor intellectual property (IP). For more than 25 years, Synopsys has been at the heart of accelerating electronics innovation with engineers around the world having used Synopsys technology to successfully design and create billions of chips and systems. The company is headquartered in Mountain View, California, and has approximately 90 offices located throughout North America, Europe, Japan, Asia and India.

1.2.2 Synopsys Design Flow of a VLSI Design

After collecting the requirements and identification of the problem, it is necessary to create a set of solutions. The chosen solution will be one that best meet the design constraints. Afterwards, start the development of the RTL code dividing the solution into blocks and defining the functional and behavioral. These specifications depend on the requirements gathered by designers in meetings with stakeholders. The design team must specify the behavior and functionality of the blocks and system.

To cover all the needs of the different stakeholders, which can become hundreds, the system must be quite reconfigurable. This aspect has to be taken into account in the course of the project. The next step in this flow is to implement the behavior of each module/block and, for each, it is necessary to validate its functionality. Completed the implementation of each module and its verification, it follows the system integration. To validate the system, the verification team can compare the behavioral model - implementation - with the functional model and/or to verify a set of corner cases. Having checked the coherence between models, behavior and functional, the project can proceed to synthesis process and then post-synthesis verification. At this point, the system is composed by primitive blocks that will be used in the place&route to create the physical organization of the system. Having a physical model of the system, the designers can check whether the timing constraints are met or not. This is a critical step in the design flow since it can force the change the implementation so that the constraints are met.

At this stage, the System's IP is completed. However, test chips are manufactured to ensure that the functionality is correct and that the requirements were met. The figure 1.1 graphically shows the company's design flow. It also shows the steps for the RTL design.

Throughout the process, it is essential to document all project stages. These documents are used by designers to understand the project and some documents are used by costumers to understand what the project does and how they can interact with it. Invariably, this task is time consuming and there is a risk that the created documentation is incorrect and/or incomplete.

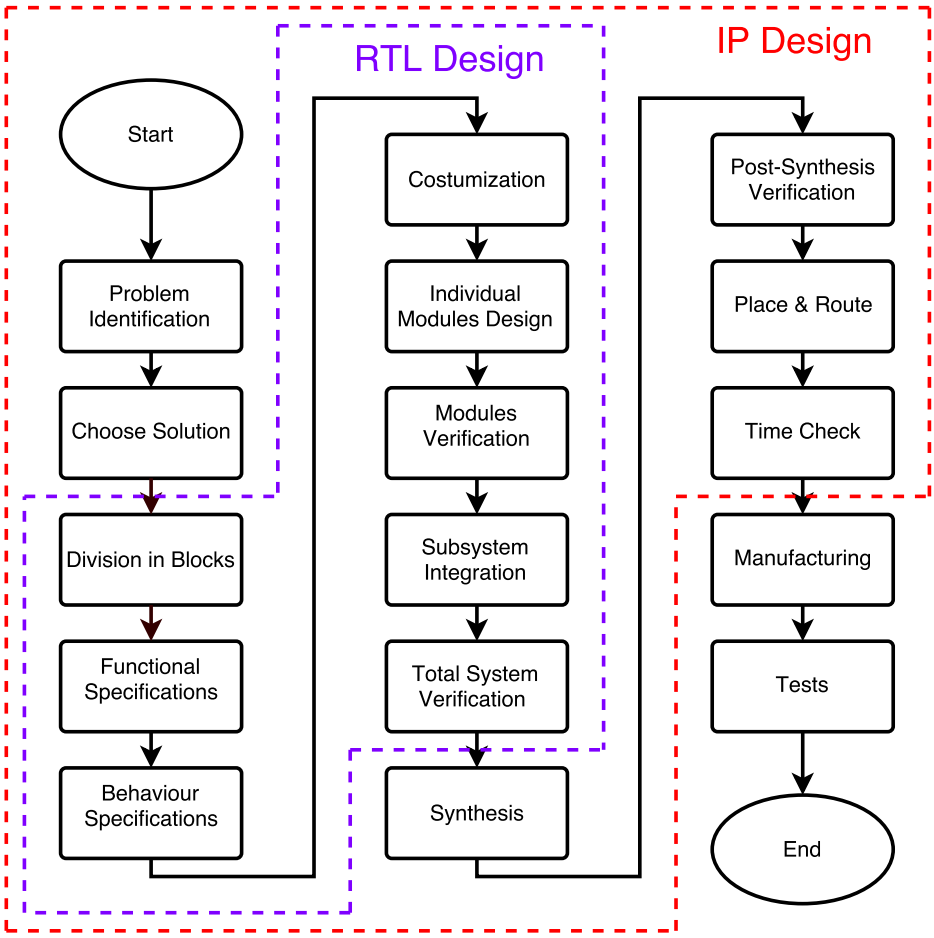

Figure 1.1: Synopsys Design Flow

One of the key steps is integration because it is a manual process, made by individuals. This fact may cause many errors, such as wrong connections between modules, errors of parameterization, wrong instantiations, and others. These errors have adverse effects to the entire project.

To reduce the time-to-market, it is important to improve the documentation and, using this information, to automate the Verilog code generation. The tool created in this dissertation uses SysML diagrams, used to document the project, to generate Verilog code automatically with the purpose of validating this approach.

1.2.3 Product Customization

Different costumers have different needs and that is why they needed custom products with specific features. For a company like Synopsys, it is important to create products widely customizable to minimize the design effort. Presently the design team makes extensive use of Verilog compilation directives to enable or disable sections of a design, thus configuring a product for the requirements of a particular customer.

A compiler directive (Macros) may be used to control the compilation of a Verilog code. The design team can control the creations of connections, instantiations, signals sizes, and more. This ability allows the system customization and the adequacy of the product to the final costumer. To understand this important stage of design flow, an example will be presented.

We will consider a N-bit adder. The following table lists the additional adder functionalities.

Table 1.1: Adder Features

Signal Available	Symbol	Comments
output cout	C_OUT	1 bit carry out fixed
input cin	C_IN	1 bit carry in fixed
output ovl	OVL	overflow flag

The graphical representation of the adder with all features (I/O pins) available is presented in the figure 1.2.

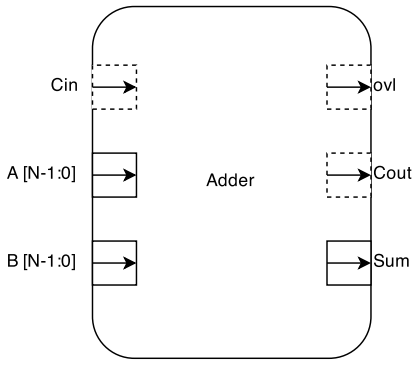

Figure 1.2: Graphical representation of adder

The correspondent code for this example is presented in appendix A. Activation of the features (Overflow, Carrie-In and Carry-Out) depend on the definition of the corresponding symbols as represented in table 1.1.

This mechanism allows to create custom products that meet individual customer needs and prevents access to features for which the license was not acquired.

1.3 Structure of the Document

This document is divided into the following chapters:

- Chapter 2 will provide some background regarding SysML as structure solution to document SoC and other engineering systems, the impact of this language in the Synopsys design flow and some tools that work with this language. Also, in this chapter, there will be addressed topics such as the hardware description languages, automatic generation tools of

hardware description language and an overview about XML. In the end, a similar work will be presented;

- Chapter 3 will be presented the generation tool of hierarchical models based on SysML diagrams. Here will also be presented the objectives of the tool as well as the design rules for correct interpretation of the diagrams. At the end, the tool workflow will be presented;
- Chapter 4 will be presented a special feature of the tool, the generation of state machine diagrams from SysML;
- Chapter 5 will be presented two case studies to demonstrate the potential of the developed tool, AutoGen;
- Chapter 6 will be presented the conclusions of this work as well as some pointers for future work;
- In appendix A will be presented Verilog code reconfigurable of an example approached in this chapter;
- Appendix B contains the complete model SysML of the second case study presented in chapter 5;
- Appendix C contains the roadmap followed in appendix B;
- Finally, in appendix D is present one of the files generated by the tool relative to the example two of chapter 5.

Chapter 2

State of the Art

This chapter presents the SysML language as a solution to the problem of technical documentation. An overview of SysML and its impact on the development of VLSI projects will also be presented, as well as some SysML modeling tools.

Another important issue is the hardware description languages with their advantages and disadvantages. In the end some tools for automatic HDL code generation will be presented, as well as some conclusions.

2.1 SysML - The Structured Solution

Systems Modeling Language (SysML) [2] is a dialect of the Unified Modeling Language (UML) for systems engineering applications. The SysML began to be developed in 2003 by the SysML Partners and it was been adopted by the Object Management Group (OMG) as OMG SysML. Presently, it is the standard for Model-Based Systems Engineering (MBSE) applications.

The SysML is a visual modeling language that supports the specification, analysis, design, verification and validation of a broad range of systems and systems-of-systems. These systems may include hardware, software, information, processes, personnel, and facilities.

2.1.1 Overview

SysML reuses a subset of UML 2 and provides additional extensions needed to address requirements in the UML for Systems Engineering.

Comparatively with UML, the SysML expresses systems engineering semantics better than UML, reducing UML's software bias and adding two new diagrams type for requirements managements - Requirements diagram – and performance analysis – Parametric diagrams. SysML is easier to learn than UML because is wider; UML is more specific for software design.

The figure 2.1 shows the organization chart of the SysML diagrams. This figure shows information about which diagrams are new or which are modified to cover the different areas of System Engineering.

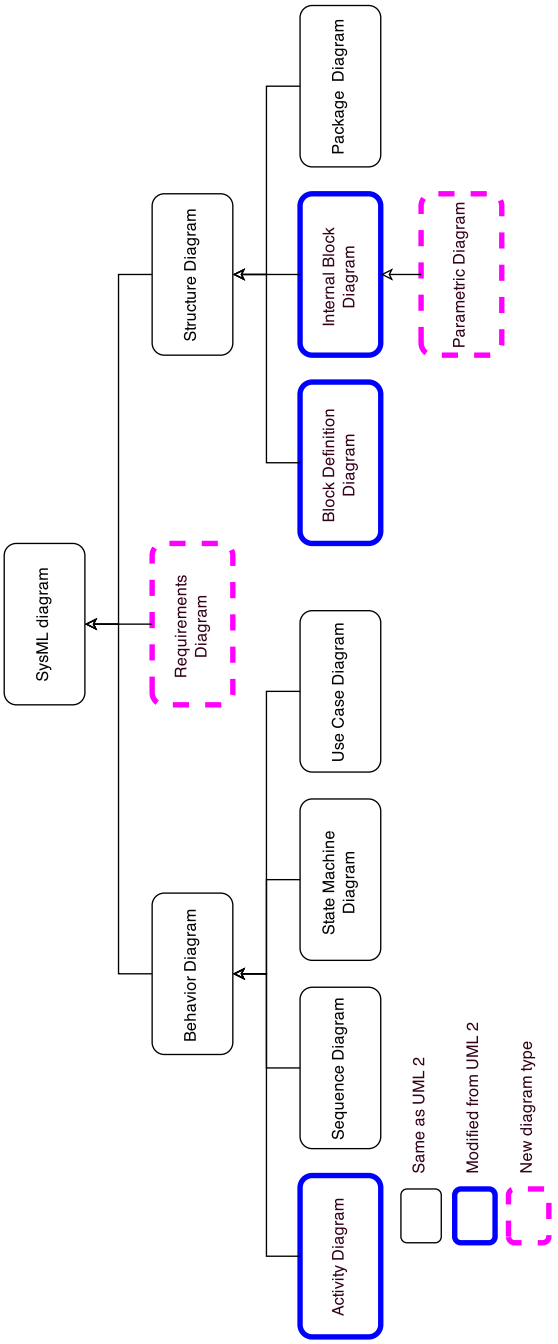

Figure 2.1: Organization Chart of SysML diagrams

The SysML settles in four main pillars: Requirements, Behavior, Structure and Parameters. Requirements diagram notation provides a way to show the relationships among requirements including constraints. It shows the relation between requirements and other model elements. Relationships among requirements can be used to define a requirements hierarchy, deriving requirements, satisfying requirements, verifying requirements, and refining requirements. All in all, the

requirement diagram provides a bridge between the typical requirements management tools and the system models. This diagram is useful for requirements validation and verification and it can be used as a communication vehicle with the stakeholders.

Parametric diagram represents constraints on system property values such as performance, reliability, and mass properties. They provide a way for specification and design models to be integrated with engineering analysis models. This diagram is useful for performance and quantitative analysis.

Structure diagram describes the system organization and it is divided in Block Definition diagram, Internal Block diagram and Package diagram. Block Definition diagram is used to show features and relationships at a high-level of abstraction, even before decisions on technology or implementation have been made. Block diagrams present blocks that can represent hardware or software or even a combined hardware/software unit. The Internal Block diagram describes with more detail what exists inside each block, e.g., attributes, parts, ports and their directions. Block and internal block diagrams are useful for system analysis and design. The Package diagram describes how a model is organized into Packages, Views and Viewpoints and it is useful for model management.

Behavior diagram describes the interaction between system elements. This pillar divides in Activity diagram, Sequence diagram, State Machine diagram and Use Case diagram. Activity diagrams show system functionality/behavior as a directed graph. These mechanisms help in the generation of functional requirements (i.e., tasks that must be supported by the system) and non-functional requirements (e.g., safety, efficiency, fault tolerance). It is useful for functional analysis. Sequence diagrams show the flow of messages between objects and it can be used for system analysis and design. A State Transition diagram is a graphic representation of the real-time (or on-line) behavior of a system. State Machine behavior can be viewed as a sequence of states in a time line in response to events. The SysML defines State Charts as State Machine model. State charts were developed for the graphical modeling of control requirements in complex reactive systems, and to overcome the limitations of basic state machine models [3]. This diagram type is useful for system design and simulation/code generation. Finally, the Use Case diagram represents the interaction between system and user or between different users and it shows system functional requirements as transactions that are meaningful to system users. This type of diagram can be used to specify functional requirements. With these diagrams, the design team can define what the system must do.

Some works are developed about modeling Embedded Systems using SysML. The authors of article [4] present the SysML and UML, and they show the limitation of UML when modeling Embedded Systems. They also show the effectiveness of SysML language. The authors illustrate this result with an example of an Embedded System: a cellular phone.

The paper [5] shows a new methodology to develop SoC/SoPC based on UML, MDD, and MARTE. In the document, it is presented a tool developed to support this new methodology. The methodology and tool are validated using an example.

In [6] it is proposed to use SysML profile for modeling SoC, with the aim to transform the

model into SystemC code. This work is a good start point for the work approached in this dissertation; however, the abstract level is different. SystemC is more high-level compared with Verilog. For this reason, it is necessary to modify this approach.

A similar work but with a different modeling language is presented in [7]. This paper suggests an approach for modeling wireless sensor networks from system specification using UML with the structure and functionality description. It is also suggested a process to generate SystemC code from UML models.

The authors of [8] present an interesting perspective about reverse engineering approach to generate SysML diagrams from VHDL. From the point of view of a company, this approach is essential to link the documentation with HDL code.

In [9], the authors present a methodology based on using UML/MARTE sequence diagrams to generate executable SystemC and VHDL code. A similar work is presented in [10].

The authors of work [11] show that it is possible to generate HDL, in this case SystemC, from State Charts. This paper shows that the Finite-State Machines can be generated from SysML models.

Despite all of these developments, there is not an efficient solution for the specific case of Synopsys. The solutions wanted are based in the generation of a non-synthesizable HDL code that simplifies the modeling approach. However, as explained in the chapter 1, the IP's development demands a different approach, because the company needs to customize the product using compiler directives and to generate synthesizable code.

2.1.2 SysML's Impact in the Synopsys Design Flow

SysML can be used to cover important stages of Synopsys Design Flow, as figure 2.2 indicated.

The implementation of the SysML in the design flow could improve the documentation of project. Some reasons are:

- Facilitate communication among various stakeholders across the System Development Life Cycle;
- Provide scalable structure for problem solving;
- Furnish rich abstractions to manage size and complexity;
- Explore multiple solutions or ideas concurrently with minimal risk;
- Detect errors and omissions early in System Development Life Cycle.

To motivate the migration for this approach, it is essential to prove that it could be used a tool to complement this documentation mechanism, allowing to connect the code with the documentation. The work [12] suggests that it is possible to generate Verilog code using SysML diagrams drawn with specific rules.

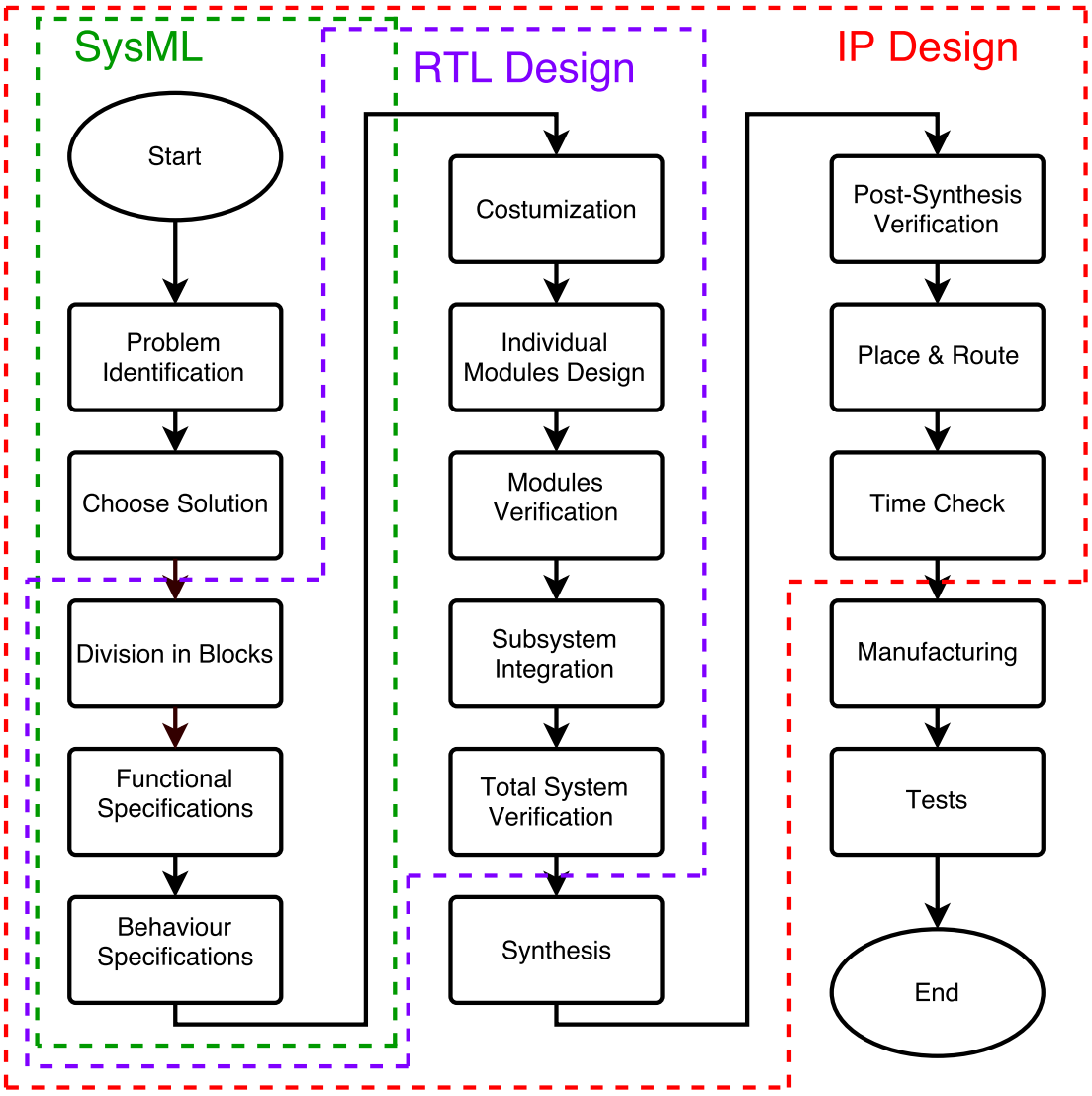

Figure 2.2: Synopsys Design Flow with SysML

2.1.3 SysML Tools

The tool suggested by Synopsys to develop the SysML models was the commercial tool Enterprise Architect [13] (EA) from Sparx. However, other tools were explored, such as the Papyrus [14] (open source), IBM Rational Rhapsody [15] (commercial) and Modelio [16] (open source). The open source versions do not allow exporting the diagram in a format that could be worked on. On the other hand, the Rhapsody is not user friendly. The IBM tool does not allow to easily instantiate a block that was created. This is an important feature because in High Description Languages, when a sub-circuit is repeated several times, a module is created with this sub-circuit and it is replicated by instantiation.

The Enterprise Architect is clearly superior to the other tools. It is the most user friendly and allows to instantiate blocks previously created. The automatic generation is a feature mentioned in

the specification of this tool and it could be used to create links between requirements and Verilog modules. Using the matrix of relations, created by EA, it is easier to analyze whether requirements were met or not. However, it is still a very recent ability and therefore it contains many gaps, e.g, limitation of characters numbers in the field that defines the portmap of instantiation, inability to redefine parameters. The solution is to redefine parameters upon instantiation using the field for input name. However, the problem of the number of characters remains. There is another gap: weak capability of error detection, inability to detect which variables were not declared, the need to use a text editor to realize where is the error encountered by the simulator is and the difficulty in creating hierarchical modules.

But, the XML files created from diagrams have all information of the model and it can be used to generate hierarchical modules improving the project development time.

2.2 Hardware Description Languages

In nowadays, the VLSI circuits are created using specific languages - Hardware Description Languages (HDL). VHDL (VHSIC Hardware Description Language) and Verilog are the most important and the most used to describe and synthetize VLSI circuits. The Synopsys uses mainly the Verilog to describe its digital circuits at the RTL level.

The table 2.1 shows a little comparison between Verilog and VHDL.

Table 2.1: Comparison between VHDL and Verilog

Feature	Verilog	VHDL
Modulating Capability	Best constructions to modeling the logic level	Best support for abstract models and delay models
Data Types	Very simple, most hardware level	Support abstract types created by the designer
Parameterization	Support only models with parameters and instantiation with parameters redefinition	Have constructions to parameterize number of bits, to replicate structures and models configuration

An HDL is different of a program language. A program language is used to describe an algorithm while the HDL describes the components of a circuit and its connections.

Such as the program languages, it is possible to work at different abstraction levels. The table 2.2 shows a comparison between abstraction level of hardware and software.

Table 2.2: Comparison between abstraction levels of Hardware and Software

Hardaware	Software
SystemVerilog/SystemC	C++
Verilog/VHDL	C

Working with high level HDL, such as SystemC and SystemVerilog, simplifies the solution description because it is closer to human language but it has the disadvantage of being more difficult to synthesize. In a VLSI project, it is important to work in a high level of abstraction but it is also synthesizable. To be able to synthesize having a high-level language, it is necessary to associate high level HDL expressions to the code structures predefined of a synthesizable description language. This approach can reduce the performance of the project increasing the area or decreasing the clock time because it might be necessary to add more resources than needed.

The main Hardware Description Language in Synopsys is Verilog 2001, which means it represents approximately 80 or 90 percent of the RTL code used. For this reason, the HDL generated by the tool developed in this work is Verilog 2001. In addition to the languages mentioned, there are others of different abstraction levels.

The authors of paper [17] present a new Hardware Description Language – Gel – that , according to them, enables quick scripting of high level design and can be easily extended to new design patterns. This language is extremely succinct and it is expression oriented. The paper names a compiler developed to translate Gel to Verilog. The work presented in [18] refers to a new abstract language (Chisel) which can generate a high speed C++ -based cycle- accurate software simulator, or low-level Verilog designed to map to either FPGAs or a standard ASIC flow for synthesis.

In the scope of this dissertation, there is some interest in using mechanisms of high level HDL to simplify the diagram in SysML, like the interface notions of SystemVerilog [19]. With this mechanism, the designer can reduce the connections necessary between modules, reducing the design complexity. This strategy will be presented in the chapter 3 to simplify the diagram connections.

2.3 Tools for Automatically Generating of HDL Code

An alternative to the modeling of a solution in a RTL level is to describe the algorithm of the solution in a high level language and, subsequently, to synthetize for a RTL model in a HDL, High Level Synthesis (HLS)[20]. These high level languages can be SystemC, C or C++. The high level code is analyzed, architecturally constrained, and scheduled to create a RTL hardware design. This approach allows working at a higher level of abstraction (algorithm level).

HDL Coder from Matlab [21] allows generating HDL from Matlab functions, Simulink models or a combination of the two. The work [22] shows the application of this tool to create a HDL code from CAN implementation. The particularity of this tool is, using a model based in blocks and specific functions of Simulink and Matlab, to easily generate HDL code, helping to minimize the complexity of project associated to RTL model design.

Other tool that generates RTL code is the Vivado System Generator from Xilinx [23]. The System Generator has blocks with a specific function and with RTL associated designs by Xilinx. With this, the designers can create a model connecting different blocks and after that, generate HDL code. This approach simplifies the generation of Digital Signals Processing for FPGAs.

The Enterprise Architect [24] allows the creation of an HDL from UML/SysML classes. For each module it is necessary to create a Verilog class. In this class, the designer can generate synchronous events (always) using synchronous methods, asynchronous events (assign) using asynchronous methods and it also allows to create events that occur at system startup (initial). As far as attributes are concerned, it is possible to define parameters, registers or wires as well as the size of them. To define the interfaces, the designer has to set the ports of the class with the name, type and size. To create the hierarchical module, it is necessary to create a new class and instantiate Verilog modules as system parts. The form of instantiation is somewhat limited both in redefinition of parameters as well as in the declaration of the ports of instantiation (portmap) because there is a limitation in characters number input. To redefine these parameters, the new settings have to be introduced during the instantiation in the field which defines the name. However, the limit of characters continues to be a problem.

These examples only generate individual models or hierarchical models without supporting compiler directives, important feature for Synopsys. This requirement together with the documentation necessity justifies the creation of a specific tool.

2.4 XML

Extensible Markup Language (XML) is a simple and very flexible text format derived from ISO 8879. This language is much like HTML. The most important difference between XML and HTML is the goal. The XML was created to describe data, with focus on what data is, while the HTML was designed to display data, with focus on how data looks. Other XML particularity is the fact that the tags are not predefined. They can be created in function finality of file. A simple description of this dissertation is presented below in XML format.

Listing 2.1: XML example

```
<dissertation>
        <title>A SysML-based Design Flow
                        for Digital VLSI Circuits</title>
        <author>Manuel Oliveira</author>
        <supervisor>
                <first>Jose Carlos Alves</first>
                <second>Antonio Pacheco</second>
                <third>Miguel Falcao</third>
        </supervisor>
</dissertation>
```

As shown in this example, it is very simple to describe the main aspects of this dissertation (title, author, and supervisors) on XML.

This markup language can be used to specify and describe hardware and, with a specific parser, generate HDL as shown in the article [25] when the authors describe a methodology for building

SoC systems having as start point XML specifications oriented to system on chip platforms with IPs integration.

This format, used in Enterprise Architect, will allow creating a specific tool to generate HDL code (Verilog) from SysML diagrams exported to XML.

2.5 Similar Works

In [26] it is presented a similar work with what will be developed in the later chapters. The goal of the author was to generate HDL code from XML files (IPXACT, a variant of the XML). In this case, the files are previously created and it describes the hardware components. With this information the tool, created during the work [26], (Automated RTL generator) generates the module.

However, the solution which will be presented in the chapter 3 and 4 is not limited to generate just one module. The focus is to generate a hierarchical model based on SysML models with a specific characteristic: using compiler directives to allow creating an extensive RTL solution that covers different profiles of customers. Other characteristic is the fact that the XML file, intermediate format of the process, is obtained from a commercial tool. This commercial tool does not follow an IPXACT standard or a form that simplifies the creation of a tool to generate HDL code. The information of diagrams is distributed by XML file, this way the searching in the file will be harder.

2.6 Conclusion

SysML is an excellent approach to create technical documentation which could help in design flow of Synopsys. However, the automatic generation of HDL code would make this modeling language more interesting. Some works show that this idea is possible, although an effective solution for Synopsys still does not exist. The tools created to generate HDL from SysML do not support compiler directives. On the other side, it is possible to work in a high level of abstraction, coming closer to human language but this approach makes the generation of HDL code for synthesis harder and ineffective, reducing performance of the IPs.

To get generating HDL code (Verilog) using SysML diagrams it is needed to design the diagrams with an approach near of the HDL. This way it is possible to generate hierarchical models of Verilog code using the information contained in XML files exported from SysML diagrams. The interfaces definition can be used to simplify the diagrams; however, when HDL code is generated, the interface has to be converted in bonds that comprise it. This approach will be explored in the next chapters.

Chapter 3

AutoGen - Automatic Generation of Verilog Code based on SysML

The tool of automatic generation of Verilog code will be presented in this chapter. The rules to modeling the VLSI project will also be presented for correct interpretation of the tool as well as the tool flow and some conclusions.

3.1 Goals of the Tool

The AutoGen, tool developed during this work, simplifies the VLSI circuit design, using the diagrams created to document the project, to generate synthesizable and configurable Verilog code. This approach allows integrate the code with documentation taking advantage of SysML's ability to systematize the design process of a block diagram and functional description, such as the dependencies between modules, data transactions and control. Unifying the system description facilitates the interpretation by various teams of the same design, such as the verification team, facilitating the development of testbenches.

Specifications of this tool are:

- Simplify connection between modules to readability of diagram using connections based on interfaces;
- Create mechanisms that allow module's parameterization;
- Support signals for preprocessing;
- Generate Verilog 2001 code to be synthesized;
- Verify the synchronization between the diagrams and the code.

Basically, as suggested in figure 3.1 this tool, to generate the hierarchical model of Verilog code, receives as input a XML file with all information about the diagrams and it generates all

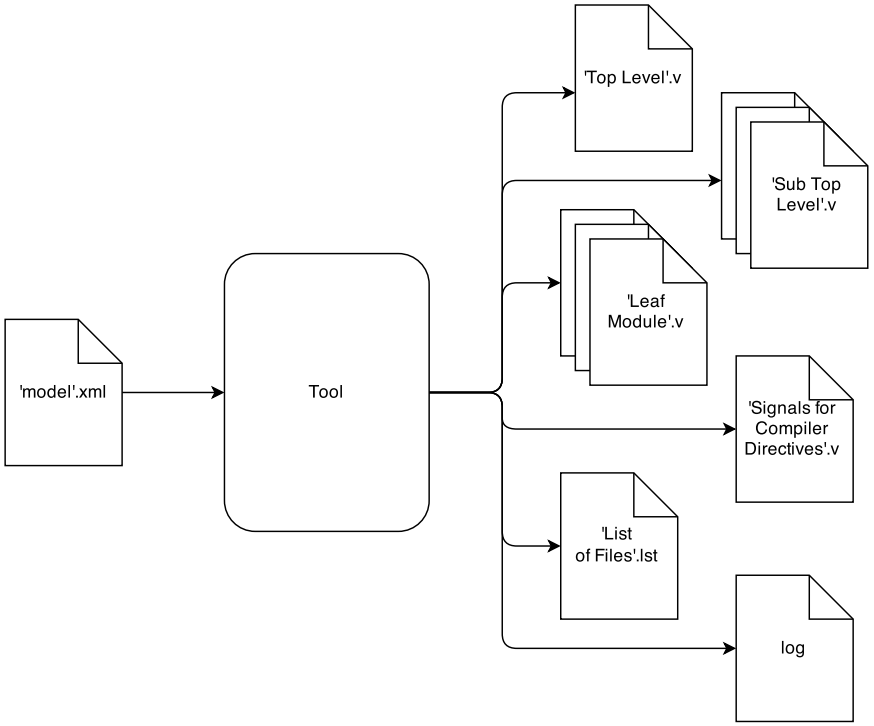

Figure 3.1: Overview about automatic generation of a hierarchical model in Verilog.

modules in Verilog, the signals for pre-processing with comments for each one, a list with all the Verilog files generated and a log file with a report about errors found.

To minimize the complexity of a project, a module can gather other models turning into a complex module. In the base of hierarchy are the leaf modules which are formed by sequential and combinational circuits. A description of this structure is represented in figure 3.2.

To allow verifying the consistency between SysML model and Verilog files, a mechanisms of revision was developed. This mechanism uses the files previously generated and the XML file exported from SysML diagrams to compare the different models and it warns the designers if a modification was found through the log file. This mechanism is represented in figure 3.3.

3.2 Design Template

To create the AutoGen, the first task was to define which diagrams would be used and its purpose. This decision was taken in accordance with the purposes of each SysML diagram. For this reason and based on section 2.1.1 of the State of the Art, it was decided to use the Blocks diagrams to define the modules of the entire project and the Internal Blocks diagram to define hierarchical structure of each block. In order to make the interfaces and the symbols for pre-processing visible in a diagram, the Blocks diagrams were indicated for this purpose. For these reasons, a template was created to maintain the diagram organized and readable. This template is shown in figure 3.4.

As shown in 3.4, the design is divided in 5 packages.

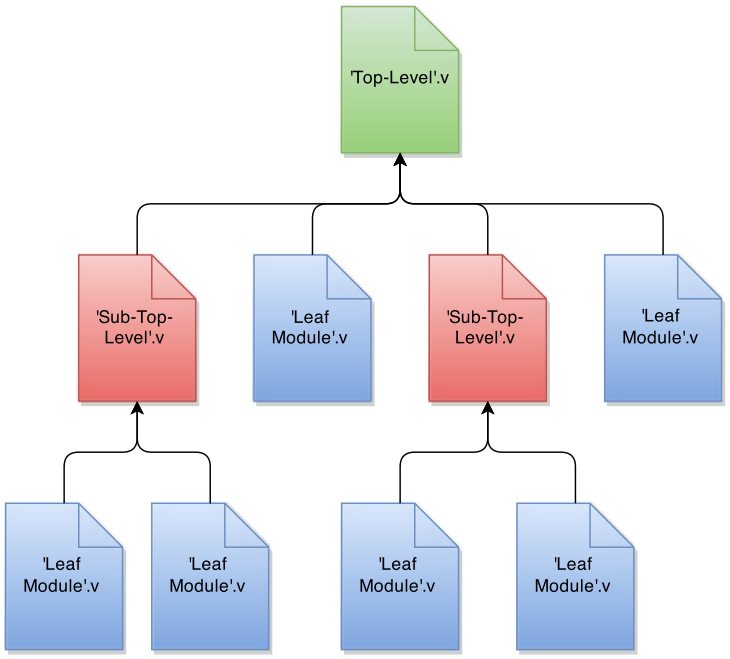

Figure 3.2: Example of a hierarchical model.

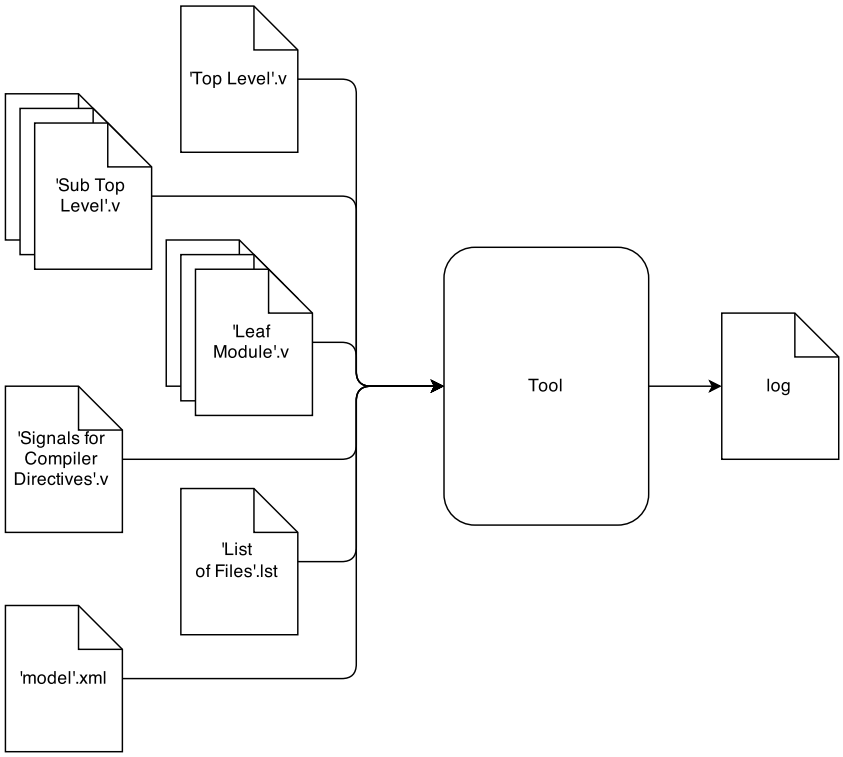

Figure 3.3: Revision mechanism.

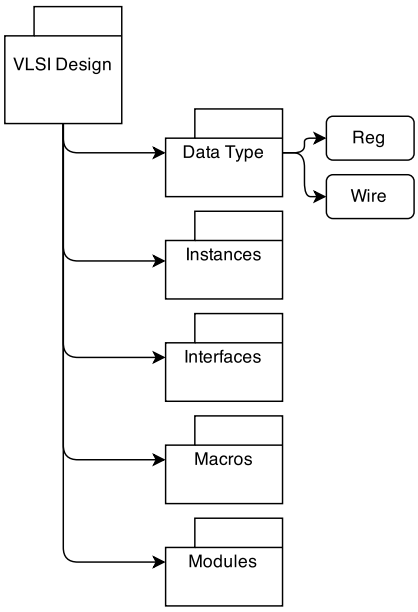

Figure 3.4: Template for the Enterprise Architect to modeling a VLSI design

1. *DataType* - The *reg* and *wire* types are not defined in the standard data types of the EA, for this reason it was necessary to create a mechanism to define them. The solution adopted was to define two *signals*, *reg* and *wire*, which can be invoked to define a port as wire or register;
2. *Instances* - In this package, all modules instantiated are saved to maintain the project organized and user friendly;
3. *Interfaces* – The interfaces created to simplify the design complexity are stored here;
4. *Macros* – The symbols for the compiler directives are defined here;
5. *Modules* – In this package, all modules defined in the project are stored.

This division simplifies tool searching process since these information will be stored in XML file according to this organization. The details of each package will be presented in the next section.

3.3 Design Rules

To create a VLSI design model in SysML, it is necessary to specify some rules.

3.3.1 Signals for preprocessing

The signals for preprocessing are used to control the compilation of Verilog code. With this mechanism, it is possible to define different implementations of the same code; solution found to

create specifics profiles of customers only by definition of symbols which will be interpreted in compilation time. The symbols can be used to define the width of signal, just assign a value to the symbol.

This definition is made in the *Macros* package. The symbol definition consists in to add a constraint block in the diagram correspondent. In the properties, the designer defines the name and the value, if it is necessary to attribute a value to the symbol. If the designer wants to comment on the symbol, for example, informing the functionality of it, he can create a note and associate this note to the symbol.

3.3.2 Interfaces

To minimize the number of connections needed in diagram, the designers can gather signals in an interface. It was explored two options for interface declaration. A first version was to create two interface definitions for a connection because it was needed to define what signals are input, output or inout and the type of signals. With this approach, each connection needed two interfaces, one at each end. To minimize the number of interfaces needed, it was created a new method to define interfaces. This new method is similar with Master-Slave notion. The idea is to associate a direction in a port that define the master, if port is output, and the slave, if port is input. This approach is based on who start the transmission and it is very usual in a SoC. To understand this concept, an example is shown in figure 3.5.

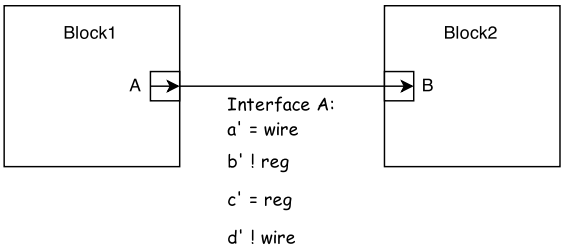

Figure 3.5: Example of an interface.

Block1 and Block2 share the same interface, *Interface A*. This interface is composed by 4 signals: $a´$, $b´$, $c´$ and $d´$; and it connects the port *A* of Block1 to the *B* of Block2. The symbols = and ! show if the direction of this port is the same of port direction (=) or different (!). For example, the signal $a´$ of Block1 will be output while for the Block2 is input. Other information is the type of signal in case of the signal to be output (input is always *wire*). The check between ports is done by the name, i.e., Block1 and Block2 have the same name for the port of the interface A and they are connected by a wire.

This mechanism not support cross connection such as Serial Port (TX connects with RX). In case of SoC, these protocols are not usual because it is more effective to connect different modules by a parallel communication and it is not necessarily more expensive. Nevertheless it was proposed a solution. This solution was the introduction of an element that would be used to

define which connections would intersect. This introduction of additional elements complicates the diagrams and its understanding.

The interfaces are defined in the *Interfaces* package. The designer puts an interface element in the diagram and he defines the name and the signals which compose them in the interface properties. The definition of signals follows the steps described previously.

3.3.3 Modules and Hierarchical Module

It was explored two approaches to define a module. The first approach is represented by the example in figure 3.6.

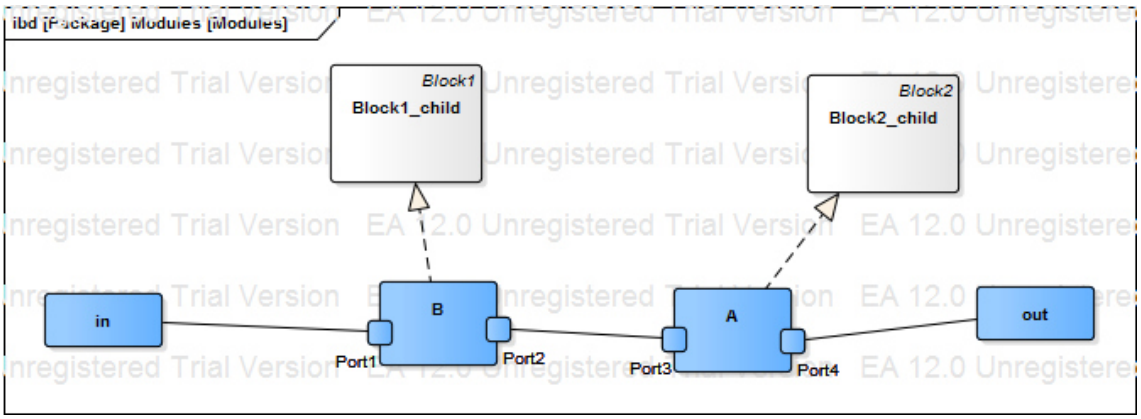

Figure 3.6: First version of module declaration.

The blue blocks, *A* and *B*, represents a instantiations of module *Block1* and *Block2*, white blocks. The signals of this TopLevel representation are indicated by blue boxes *in* and *out*. This approach is not useful to document the project and, by this reason, a new approach was developed.

In this new approach the modules are all defined in Blocks diagram of *Modules* package. For each modules, the designer sets the name, parameters and constraints (symbols for compilar) in properties. The signals for outside communication are defined as *Flow Port*. In the properties of port, it is possible to define de direction, width, type and constraints, if they exist. In this diagram, the designer defines all modules that compose the entire project.

In order to enhance the ability to completely define a digital system using a SysML model, it was created a mechanism to define a block of Verilog code as a method of a block, approach based on object-oriented languages. With this mecanism the tool generates not only the hierarchical structure of a model but also whole system, i.e., structure and behavior.

To create a hierarchical model, the designer only has to create a diagram of internal blocks (new child diagram) and instantiate the necessary modules. A representation of this model is represented in the figures 3.7 and 3.8.

With this approach, the diagrams can be used not only to generate code but may also be used to document the project and it is the best way found to simplifies the generation of a hierarchical model.

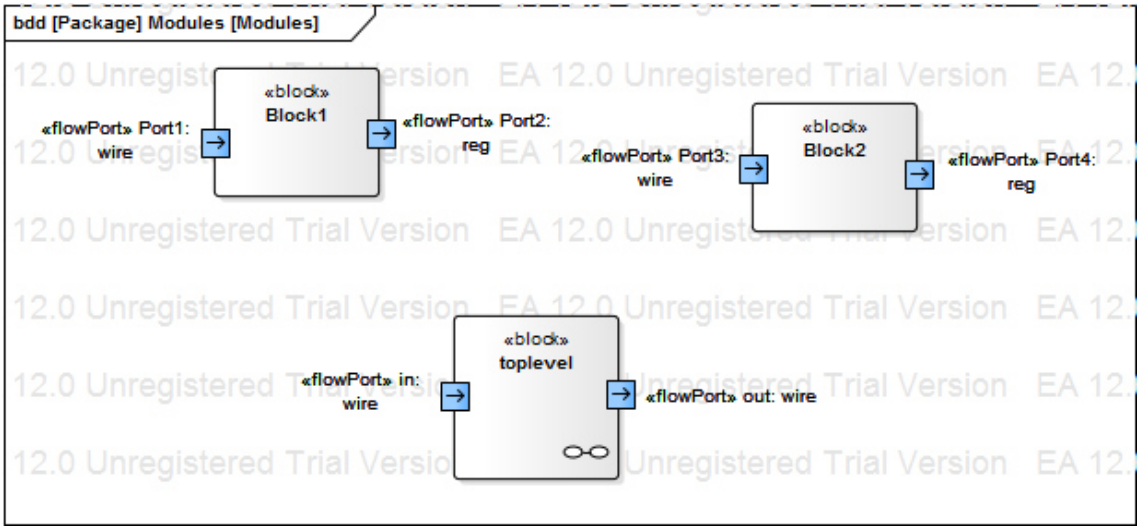

Figure 3.7: Example of modules declaration.

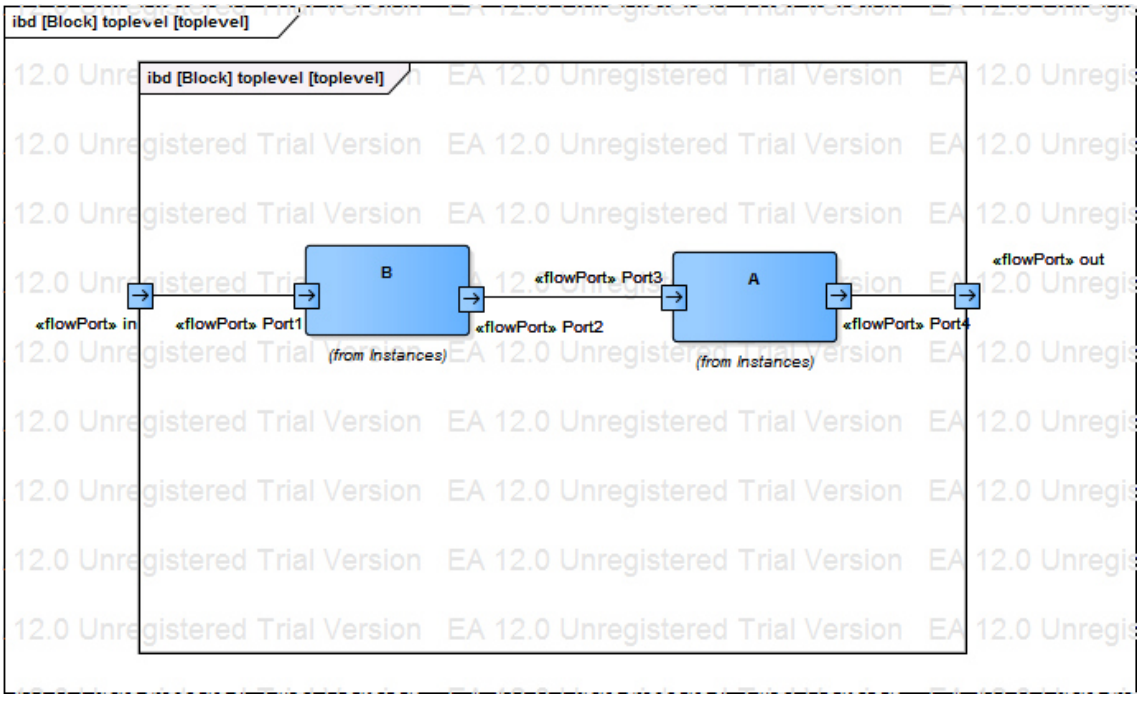

Figure 3.8: Example of a top level module.

3.4 Flow of the Tool

The AutoGen is divided in two structures, the structure which generate Verilog code and the structure which verify the consistency between the model in SysML and the files generated. However, these structures starting with the same steps. The figure 3.9 represents a overview of the tool.

The tool starts by reading the XML file where capture the elements that will be used to generate Verilog code.

The first step is capturing the symbols that will be used to activate the implementation of

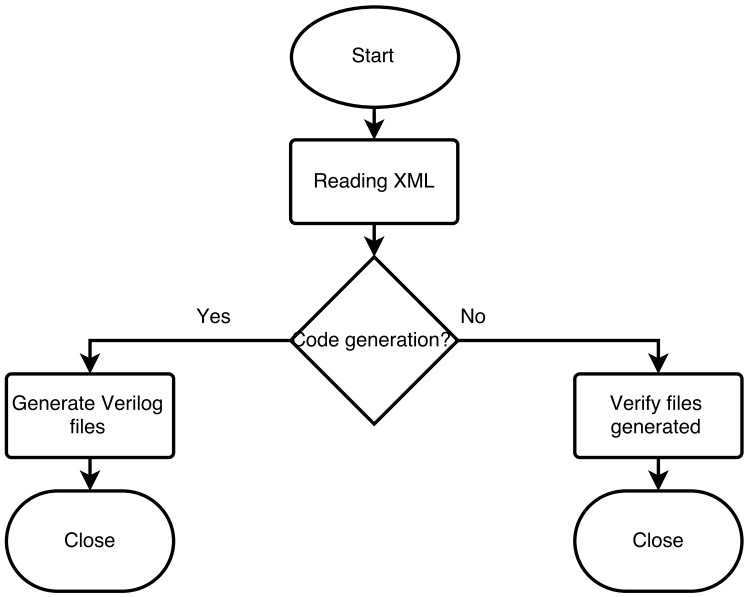

Figure 3.9: Overview of Tool Flowchart.

features. The next step is to capture the data types, i.e. to find the ID of *reg* and *wire*. As previously indicated, the *reg* and *wire* types are not a primitive types of the Enterprise Architect. Before modules capture, it is necessary to capture the information about interfaces (information about signals, width and type, and constraints of the interface, dependency of symbols). The interfaces operate as a port type and, because of that, this information has to be known before of the modules to allow define correctly its ports. The following step is to capture the modules information (name, types and width of ports, parameters, constraints and Verilog code like methods of block). The association between modules and constraints, types or interfaces is done by internals ID of the Enterprise Architect that are referenced in XML file. Known the previously elements, it is time to find the modules instances. Typically, the tool does a copy of all structure of module associated to the instance and it changes the specific elements that characterize the instance (name and some parameters which may be different of default value). The next step of this flow is to associate the instances to the modules. As in the case of the ports, this association is done by IDs. After that, the comments for preprocessing symbols are captured and associated to the correspondent symbol. To finish the XML reading, the connections inside of a hierarchical model are captured. In the SysML model can exist a set of connections that represent the same point in the hardware, and, for this reason, these connections represents the same wire.

To resolve this issue an algorithm was implemented. The first step of this algorithm is identify the connections that are directly linked with the external ports of module and, in this case, the connection receives the same name of port. This approach minimizes the number of wires required to be declared. The next step is to find the connections that are connected with same point of the connections that already received a name. The connection that are not connected with

external ports directly or indirectly receive a generic name (w_number, in the case of interfaces - w_number_signalName). The example of figure 3.10 will be used explain better the algorithm.

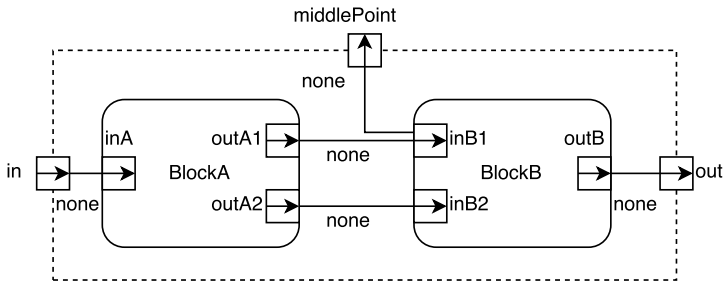

Figure 3.10: Example where the algorithm will be demonstrated.

The figure 3.11 represents the first two steps of the algorithm. In the first, the connection receives the same name of external port and in the second, the connections without name receives a generic name.

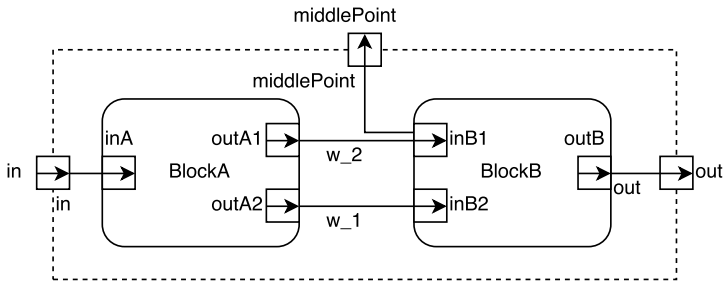

Figure 3.11: First two steps of algorithm.

The third step consists in to find connections that represents the same point in hardware. The name that remains is what is associated with an external port. If this does not happen or if both ports are associated to external ports, then the choice will be random. This final step is represented in figure 3.12.

This first phase of the tool is represented in figure 3.13 with all steps described previously.

The next phase depends of the goal, code generation or revision of models. Starting by code generation.

After reading the XML file, the AutoGen has a internal representation of model based on a combination of dictionaries, vectors and classes. These structures will be used to generate the Verilog code. This phase follows the steps described in figure 3.14.

The first file created is a file with all symbols used to activate the features defined. After that, the tool starts writing the modules. The first part of this process is to write the header of module with parameters and ports. The write style follows the design guideline of Synopsys. However,

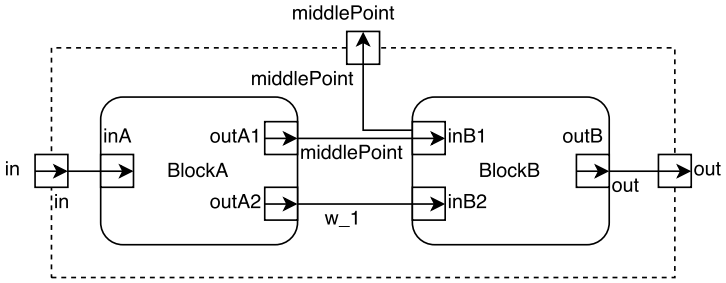

Figure 3.12: Third step of algorithm.

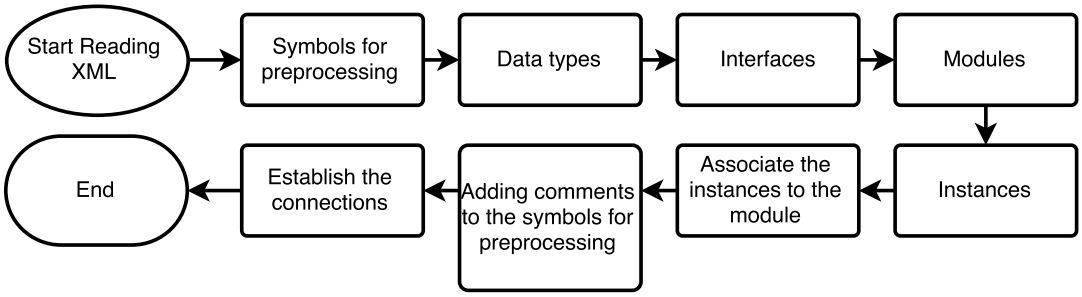

Figure 3.13: First phase of the tool.

during the time development, a problem appeared. The last port declaration has not comma but, with the compiler directives, a problem can appear. The next example will illustrate it.

```
module adder (
    input   wire  [N-1:0]  A ,
    input   wire  [N-1:0]  B ,
    output  wire  [N-1:0]  Sum ,
  `ifdef C_OUT
    output  wire  Cout
  `endif
  );
```

In this example, if *C_OUT* is not declared, the compiler will generate an error in the comma of *Sum* line. The solution is to insert the comma before the port declaration with the exception of the first line which has not comma. This approach is a simple solution that resolves the problem described. The example above with this new solution is presented below.

```
module adder (
     input   wire  [N-1:0]  A
    ,input   wire  [N-1:0]  B
    ,output  wire  [N-1:0]  Sum
```

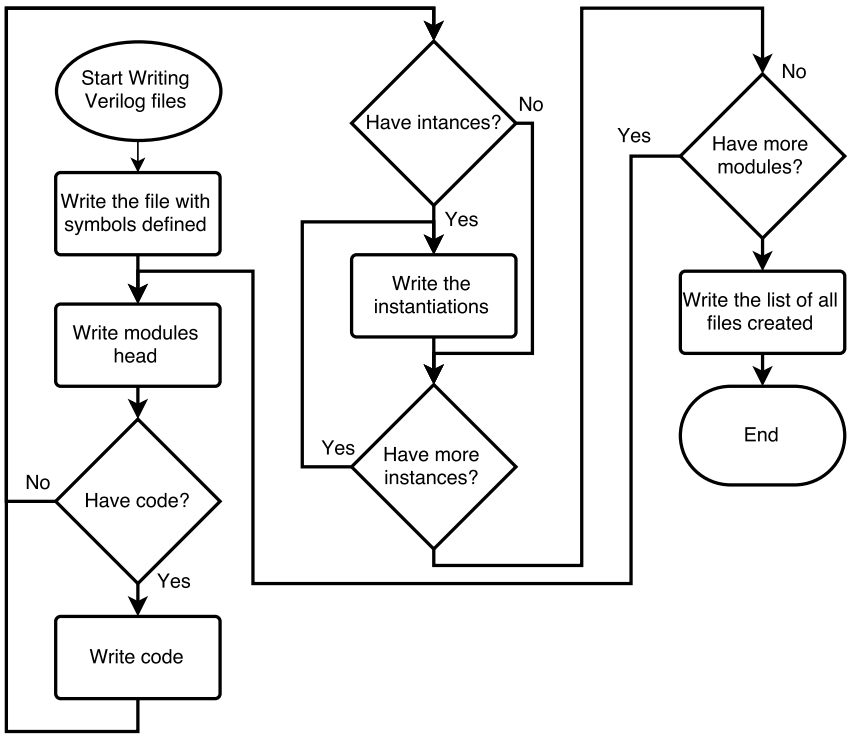

Figure 3.14: Code generation phase.

```
`ifdef C_OUT
  ,output  wire  Cout
`endif
);
```

After of the header, the AutoGen will write the Verilog code associated with the module and which has been declared as block's method.

Subsequently, the tool writes the instances which are associated to the module following the SysML model. The instantiation is done following the dependencies of SysML model, i.e. if an instance or a port requires a symbol declaration; this will be shown in the code.

A file with all Verilog files generated is created in the end of tool. This file can be used to inform the compiler about which files are part of the project.

During the development, the designers may have to make changes in the hardware description language code. These changes may lead to loss of correlation between the code and the model. To verify this agreement was implemented in the AutoGen a verification mechanism.

The implemented mechanism follows a flowchart similar to the code generation engine. The tool checks the generated code based on the way it was written previously, that is, taking the block of information that would be written to the file, it uses it to search in the file its existence. If there is no match, the tool generates an alert. The verification is not performed to the entire file at once but rather by parts, i.e., first scan is performed at the file header, starting with the name

and parameters followed ports. In case of missing some parameter or port, the tool will inform its absence, the same for the case of new parameters or ports. They are then checked if the existing wires in the code coincide with those that had been generated previously. Should there be more or fewer wires the tool generates an error. Then it is checked if the code inserted in the model is contained in the file. The next elements to be checked are the instances. These follow a similar process to the header verification with the exception that they are not checked all parameters but those who should be re-parameterized. As with headers, will also be generated alerts in the event of inconsistency between the code and what was expected. All of these errors are reported in the log file, the console only an alert appears and the number of errors. The steps are represented in figure 3.15

3.5 Conclusion

With this tool it is possible to connect the documentation with code generation decreasing development time ensuring that the documentation is always up to date.

In the next chapter it will explored a new feature that complements the AutoGen through the use of state charts to generate Verilog code automatically.

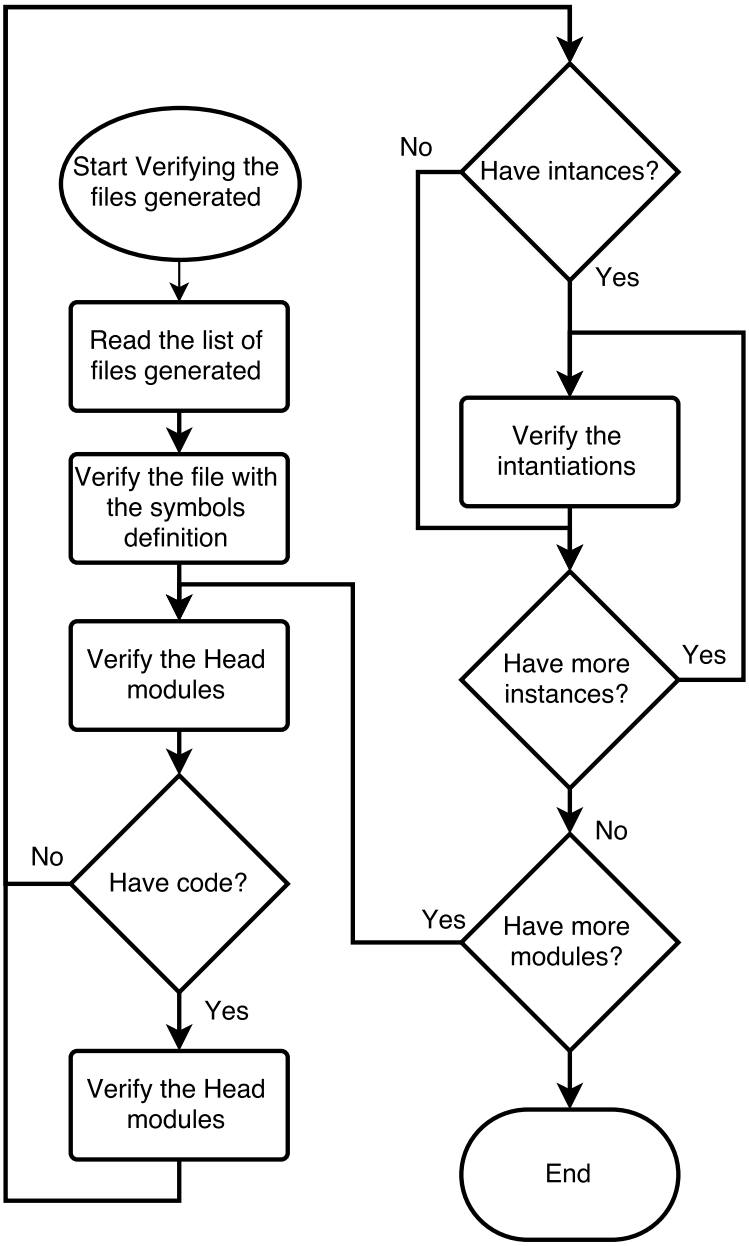

Figure 3.15: Consistency check between the model and the code.

Chapter 4

Automatic Generation FSM in Verilog Code

The introduction of the ability to generate code from the representation of state machines allows the designer up abstracting of the problem while documenting the solution.

In this chapter it will be presented the rules for drawing diagrams and the tool's approach to process this information. At the end of the chapter it will be presented some conclusions and limitations of the tool.

4.1 Automatic Generate FSM

The finite state machines are one of the mechanisms used to model data control systems. Typically, a complex digital system can be divided into two fundamental parts:

- Data path - System that processes the data;
- FSM - Controls the processing of data.

Using these modeling mechanisms to generate the Verilog code improves the developing time and ensures that the model is properly coded, preventing the introduction of errors due to bad encoding.

The finite state machines are divided into two main models: Moore and Mealy. The model of Moore only depend on the current state, while the Mealy model depends on the current state and the inputs. For lack of time, it was decided to implement the model of Moore to be easier to code in Verilog. However, once it was created a method for adding Verilog code in SysML model, it is possible to encode Mealy machine with some modifications. Encoding is performed similarly to the Moore machine with the exception of the outputs. Due to the difficulty in defining graphically the outputs in this type of machines, these will be defined by introducing Verilog code.

4.2 Draw Rules

For proper implementation of the state machine it is necessary to respect some rules. These rules are listed below.

4.2.1 Triggers

Typically, in a state machine, to transition between states is carried out at each trigger of the clock signal and after validating the transition condition. However there may be other signals to generate asynchronous state transitions, for example, an asynchronous reset. Depending on the development team or designer, reset and the clock signal may have various names, for example, the clock signal can be called clock or CLK. For this reason, it is necessary to define the name of the trigger signals.

To set the trigger signals is necessary to use the *trigger* elements. On the signal name, the designer can set the edge of the trigger signal such as in Verilog, that is, if the clock signal is activated by the positive edge then the name will be *posedge clock*. In the *trigger* properties it is possible to set what happens when the trigger signal occurs, for example in the reset.

4.2.2 States

An important aspect in the encoding of the state machine in the Synopsys is that the output is typically one register with variable size. The size is defined by the number of elements to control, i.e. each bit or set of bits in the output register activates one or more elements of the data path, for example a register or multiplexer. To simplify the diagram, the state, it is only represented the signals which has logic level one. In this way, the signal is defined as a state condition, simplifying its coding. State output signals are defined in state property. A starting point is connected to the initial state for their identification.

For the implementation of Mealy machines, the outputs are defined by introducing code into the module (such as module operation) since, in this type of machines, the graphical coding of outputs can be extremely complicated.

4.2.3 Transitions

Transitions indicate the flow of states and may or not be subject to transition conditions. In the absence of conditions, typically, the transition occurs after a trigger of the clock signal.

In SysML, the transition conditions are defined in the properties of transition. In the properties the designer can set the polarity of a signal (zero or one) or it can compare the input signal with a specific value.

4.2.4 Other aspects

To support state machines, it was necessary to modify the definition of the blocks / modules to be possible to set local parameters. For this, it was necessary to modify the parameters setting.

The local parameters may be used to define values which will be compared with the entries, for example, to set the register that will be changed. This modification allows the designer defining local parameters, records and even wires.

Regarding the states, the process is automatic. Both the register which stores the next state as the current state are set automatically. The required size for these registers is calculated based on the number of states automatically. The encoding of the states is performed as follows: the initial state is always encoded with the zero value and the others at random. For lack of time was not implemented any optimization algorithm for the coding of states.

4.3 Flow of the Automatic Generation of FSM

To support this new functionality, the AutoGen workflow had to be modified. In the process of reading the XML file, in addition to information that the AutoGen has already been collecting about the modules (parameters and ports), it will now collect others information such as, local parameters, registers and wires that will be used by the state machine. Then, a search is performed on the file to collect the remaining elements needed to define a finite state machine (triggers, states, transitions, and transition conditions). The existence of all the signals used in the state machine in the module statement, such as parameters and input and output signals, is verified. This process is performed to ensure that the state machine does not rely on signals that do not exist in the module declaration and to ensure that any local parameters of the FSM were properly declared.

The process of writing the files is identical to that described in chapter 3. The differences are due to the introduction of mechanisms that allow the AutogGen write the code from the diagram of FSM. The steps followed by the tool is shown in Figure 4.1.

If the module contains a FSM, the tool begins by declaring the parameters, registers and wires of the state machine. The parameters of state machines usually concern the codification of state names or registers that will be used to compare with input signals and, therefore, are defined as local parameters. In this phase they are also declared the registers that store information on the current state and the next state.

The next step is the block writing that calculates the next state based on the current state and outputs. This block is created based on information from current state, transitions and transition conditions associated with it.

The last step to generate state machine is the definition of output and this is where the model of Moore and Mealy are distinguished. In the model of Moore, the outputs are associated to the states and they are easily coded based on a table. The Mealy model is more complicated since the outputs depend on the current state of the system and the inputs, and, for this, it is more difficult to encode. To demonstrate the ease of coding the outputs of Moore's machine it will be presented an example next.

The tablela 4.1 is an example of the coding of the outputs of a Moore machine. The corresponding Verilog code is:

```
assign Y1 = ( STATE == Ready) || ( STATE == Run);
```

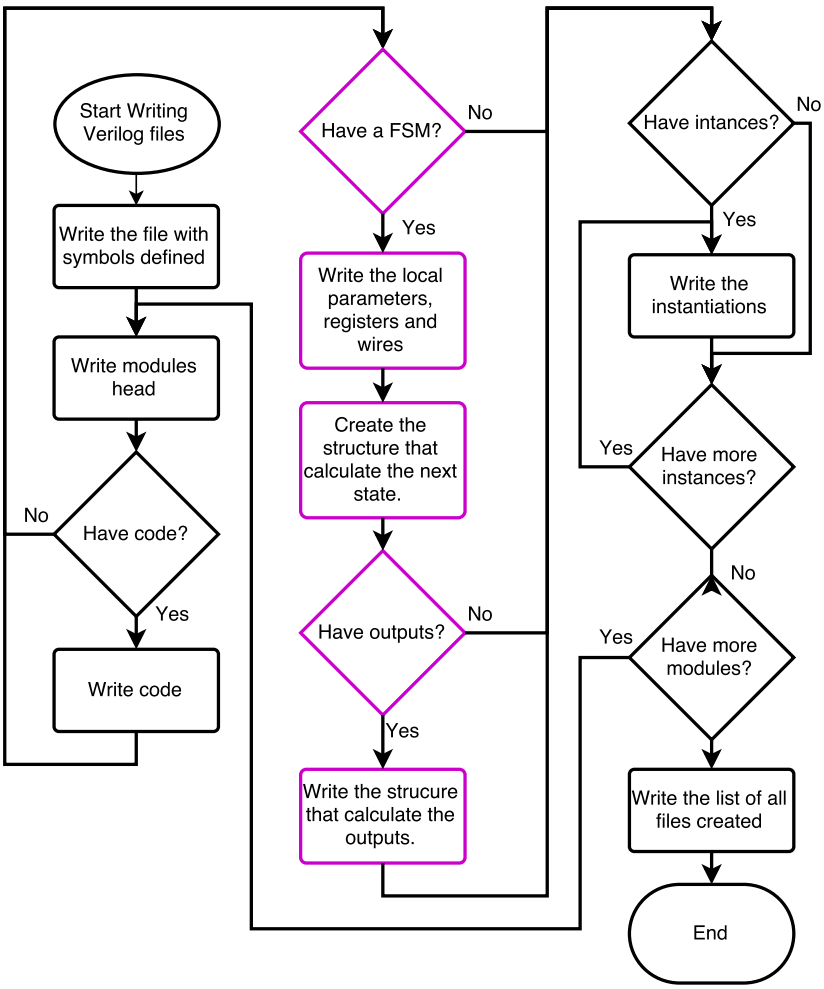

Figure 4.1: Generation of Verilog code with FSM feature.

Table 4.1: Example of a truth table of a Moore machine.

Current State	Y1	Y2	Y3
Idle	0	0	1
Ready	1	1	0
Run	1	0	0

```
assign Y2 = ( STATE == Ready);
assign Y3 = ( STATE == Idle);
```

However, by introducing code in Verilog model it is possible to encode the outputs of the Mealy machine. Thus, to create this kind of machines, the designer can encode the states using the State Machine diagrams but the outputs has to be define by Verilog code. The remaining steps of the tool are similar to those already mentioned in the previous chapter.

The verification code feature also had to be modified to support verification of the state machines. The figure 4.2 shows the changes made in the AutoGen's workflow to support this new feature.

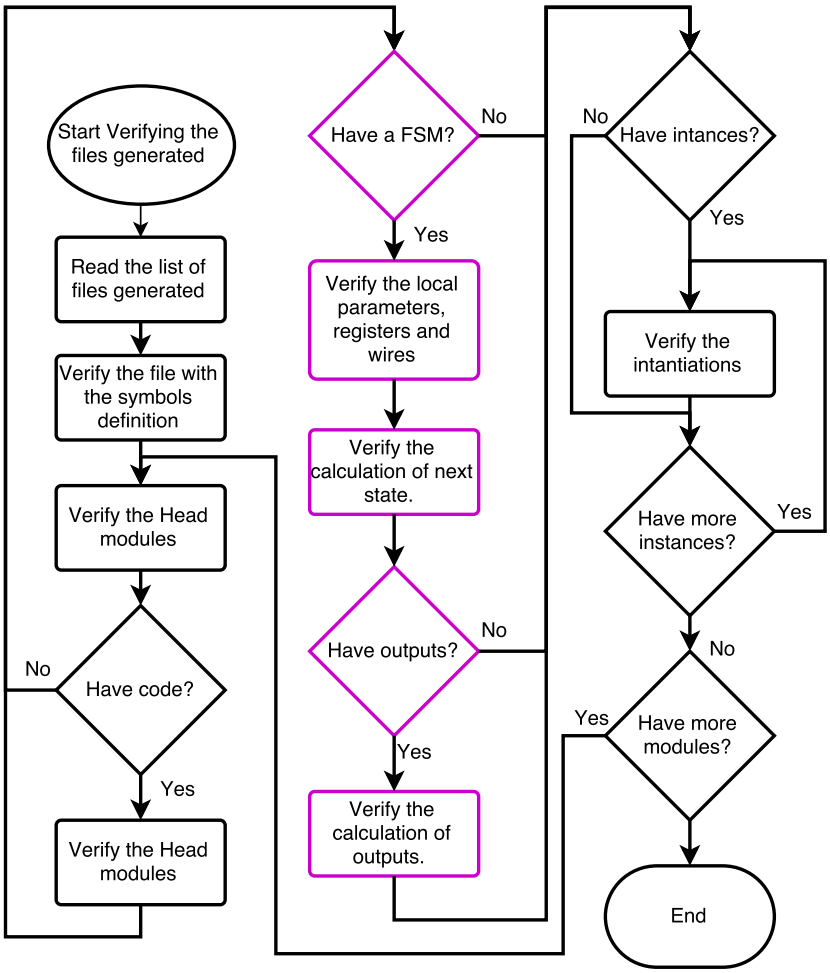

Figure 4.2: Code revision with FSM feature.

The steps taken to verify the coherence between the SysML model and the code are at all similar to the steps to generate a hierarchical model in Verilog (chapter 3).

4.4 Conclusion

Automatic generation of state machines from templates in SysML is a process that needs time to be explored all the possibilities. The model presented, and which will be illustrated with an example in the next chapter, it was created based on the simplicity and transparency, that is, the Enterprise's ability to push information from SysML's diagram to the XML file. Even so, the solution presented, along with the introduction of code in Verilog model, allows to generate any kind of state machines.

The biggest limitation of this feature is that can only be generated a state machine by module, using the diagrams to encode the state machines. This limitation can be circumvented of two ways, or encoding of the second machine is made manually by introducing Verilog code for this purpose in the model, or the module is divided into other smaller where each contains only one state machine. This last approach is always true because a module can always be divided into data path and control structure (FSM).

Chapter 5

Case Study

In this chapter, it will be presented two case studies to demonstrate the AutoGen capabilities. The first case study is an Adder. In this example, it will be explored the customization capabilities of the product and the mechanism created to modeling the problem of customization. The second case study is a more complex subsystem - DMA. With this second example, the interface mechanisms and the approach of finite state machines will be explored. These two examples follow all the rules presented in the chapters 3 and 4. In either case the code generated through the tool will also be presented. In the end of the section will be presented some conclusions about the results obtained.

5.1 Adder

An Adder is a digital circuit which performs numbers addition. The Adder is a key element in arithmetic operations but not only. This circuit is present in virtually every digital integrated circuits subsystem.

5.1.1 Specifications

The base version of the N-bits Adder is the sum of two parts without carry-out, carry-in and overflow detector. Each of these inputs/outputs consist of extra features and, therefore, will depend upon the definition of specific symbols.

5.1.2 SysML Model

As explained in the previous section 5.1.1, each extra functionality will have a symbol assigned . Figure 5.1 illustrates the symbols and the corresponding functionality.

Observing the diagram, it is possible to conclude that the definition of *C_IN* will allow to the Adder supporting a *Cin* input. Each of these symbols will be used, for example, to create dependencies between the symbol definition and the existence of ports, modules and even implementations.

The next step is to divide the problem into modules. To show a small hierarchical model, the sample will be divided into two blocks:

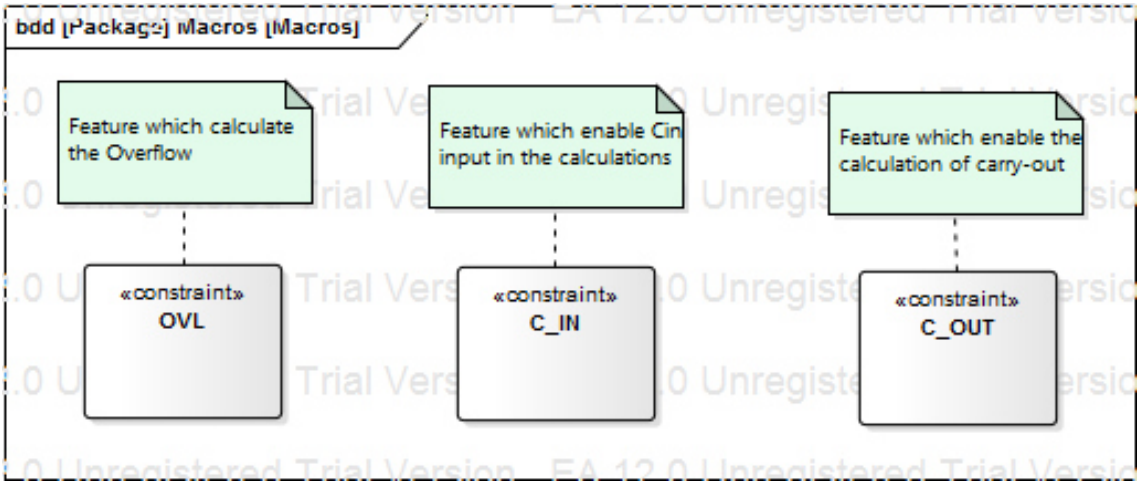

Figure 5.1: Adder Macros.

- Block that calculates the existence of overflow;
- Block that calculates the result with or without carry-out and/or carry-in.

These blocks are represented in the figure 5.2 along with the top-level module.

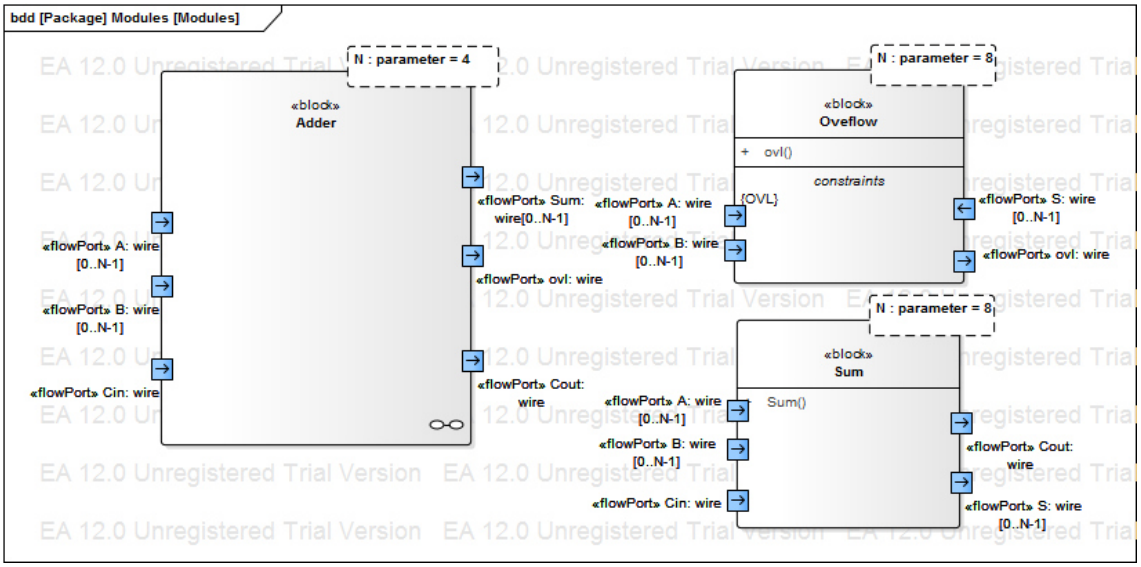

Figure 5.2: Adder modules.

It is not visible in the diagram the association between symbols and ports, is only visible the association between the modules and symbols as illustrated by the overflow identification module. However, to ensure that the generated code does not generate alerts, it is necessary to assign the correct symbol to the ports that will not result in disconnected wires.

Observing the diagram, it is readily perceptible that the modules are configurable and that the parameter is associated to the size of the ports. Also it can see that the module *Adder* has associated a diagram, in this case an internal blocks diagram.

The code that defines the behavior of the modules *Overflow* and *Sum* is represented as method of each blocks. Now it need to define how modules are connected with each other to complete the model. This interconnection is represented in figure 5.3.

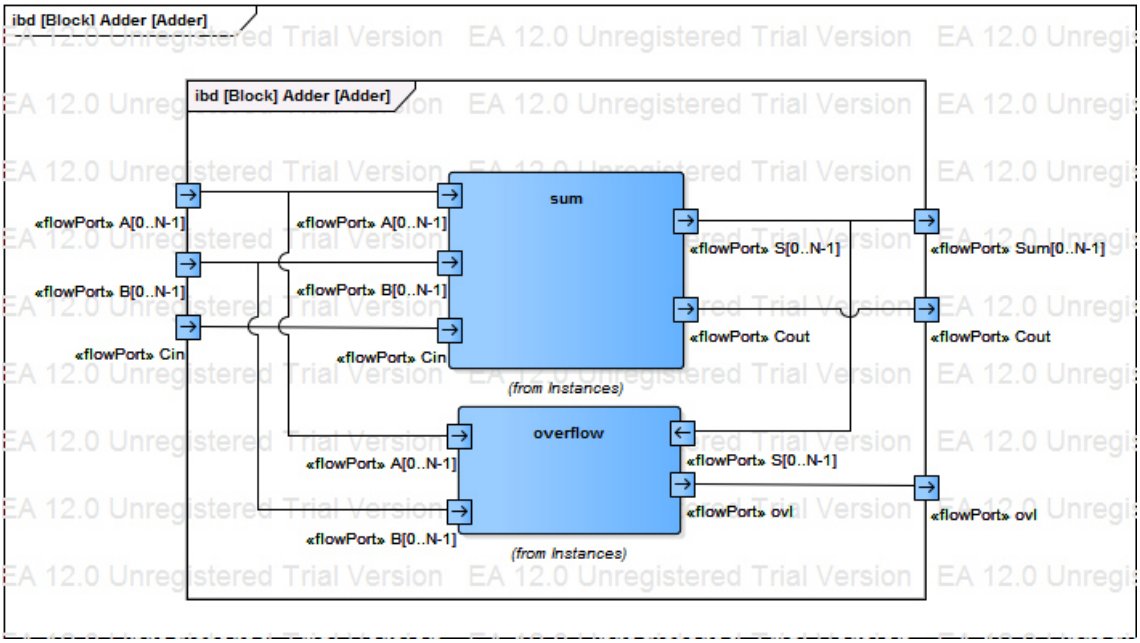

Figure 5.3: Top-level module.

This diagram shows how the different instances are connected and how to link these with top-level module, in this case *Adder*. This diagram is essential for understanding the structure of modular solution, this is a similar approach to the Verilog.

These different diagrams presented have several goals. On the one hand, they can be used to document the whole project, at the same time can help the different elements that compose the development team to understand the structure and implementation of the system. Following the rules proposed in this dissertation, the diagrams can be also used to generate all the code of the system, using the tool developed - AutoGen-, ensuring that there is a clear correspondence between the model and the resulting code.

5.1.3 Code Generated

The code generated by the tool for this example will be presented here. Starting with the file with definition of the symbols.

```
//##############################################################################
//#### Signals for preprocessing

//##############################################################################
//## Feature which enable Cin input in the calculations
`define C_IN
```

```
//#############################################################################
//## Feature which calculate the Overflow
`define OVL

//#############################################################################
//## Feature which enable the calculation of carry-out
`define C_OUT
```

The notes present in the diagram which defines the macros are copied to the Verilog file in the form of comments.

Next files to be presented will be the corresponding code of the leaf modules. The code obtained from the *Sum* module is:

```
module Sum #(
  //---- PARAMETER DECLARATION ---------------------------------
  parameter N = 8
  )
  (//---- PORT DECLARATION ----------------------------------------
     input  wire  [N-1:0]  A
    ,input  wire  [N-1:0]  B
    ,output  wire  [N-1:0]  S
  `ifdef C_IN
    ,input  wire  Cin
  `endif
  `ifdef C_OUT
    ,output  wire  Cout
  `endif
  );

`ifdef C_OUT
  `ifdef C_IN
    assign {Cout , S} = A + B + Cin;
  `else
    assign {Cout , S} = A + B;
  `endif
`else
  `ifdef C_IN
    assign S = A + B + Cin;
  `else
    assign S = A + B;
```

```
  `endif
`endif

endmodule
```

In this code it can easily understand the dependence between some ports, *Cin* and *Cout*, and their symbols. In practice, after the compiled code, these ports are accessible if the corresponding symbols are defined, otherwise the ports will not be accessible as well as the behavioral code of the same. The following code relates to the *Overflow* module.

```
module Oveflow #(
  //---- PARAMETER DECLARATION ---------------------------------
  parameter N = 8
  )
  (//---- PORT DECLARATION ----------------------------------------
     input  wire  [N-1:0]  A
    ,input  wire  [N-1:0]  S
    ,input  wire  [N-1:0]  B
    ,output  wire  ovl
  );

assign ovl = ( A[N-1] == B[N-1] ) && ( A[N-1] != S[N-1] );

endmodule
```

Then it is presented the top-level file. In this file are presented the dependencies between symbols and ports but also with the instances. It is also present in this example the redefinition of parameters upon instantiation.

```
module Adder #(
  //---- PARAMETER DECLARATION ---------------------------------
  parameter N = 4
  )
  (//---- PORT DECLARATION ----------------------------------------
     input  wire  [N-1:0]  A
    ,input  wire  [N-1:0]  B
    ,output  wire  [N-1:0]  Sum
  `ifdef C_OUT
    ,output  wire  Cout
  `endif
  `ifdef C_IN
    ,input  wire  Cin
```

```
  `endif
  `ifdef OVL
    ,output  wire  ovl
  `endif
  );

  Sum #(
.N (4)
) sum (
  .A(A)
,.S(Sum)
,.B(B)
`ifdef C_IN
,.Cin(Cin)
`endif
`ifdef C_OUT
,.Cout(Cout)
`endif
  );

  `ifdef OVL
  Oveflow #(
.N (4)
) overflow (
  .A(A)
,.S(Sum)
,.B(B)
,.ovl(ovl)
  );
  `endif

endmodule
```

All of these Verilog files were generated from the above SysML diagrams presented. The next example is more structurally complex. The FSM and interfaces mechanism will be explored in that example.

5.2 DMA subsystem

The DMA, Direct Memory Access, allows to transfer data between memories without using the CPU, freeing the CPU to perform other operations while elapsing the data transfer in parallel.

Typically, an interrupt is generated to the CPU at the end of the operation.

5.2.1 Specifications

The case study that will be presented is a simple PCI express DMA (eDMA) which only performs the function of writing - transferring of data blocks from local memory to the remote memory. The eDMA will have a simple configuration interface with signals described below:

- Two bits to define the configuration operation -reading and writing of the configuration registers;
- One signal to define the register to configure;
- One signal to send data to the registers;
- One signal to access to the data from the registers.

The eDMA will have four main 32 bits registers:

- *Xfersize* - defines the transfer size which can vary between one byte and four Gigabytes;
- *SAR* - defines the source address;
- *DAR* - defines the destination address;
- *Doorbell* - indicates the beginning of transmission.

The interface with the memories will be composed by six signals:

- *HV* - validates the type of request that will be sent to the memory;
- *Address* - line where the memory address will be sent;
- *TypeData* - line where the type of request will be sent;
- *DV* - validates the data line;
- *Data* - line where the TLP packets will be sent;
- *EOT* - marks the last bit of *Data*.

Data transmitted between memories will use TLP packets.

After the correctly configuration of the eDMA and activation of *Doorbell*, eDMA will start generating one or more TLP packets, depending of *Xfersize* value, to make requests to the local memory. The local memory responds with a Completion with data to be transferred. The eDMA will convert this Completion in a Write request to be sending to the external memory.

5.2.2 SysML Model

The appendix B shows how the design team can use the SysML language to document this example. In this section the main focus will be the diagrams that describe the system structure and behavior, following the rules presented in chapters 3 and 4.

This being an example with some complexity, it makes sense to try to group signals to minimize the number of connections represented in the diagram. In this sense, the signals were grouped to form four interfaces that are represented in figure 5.4.

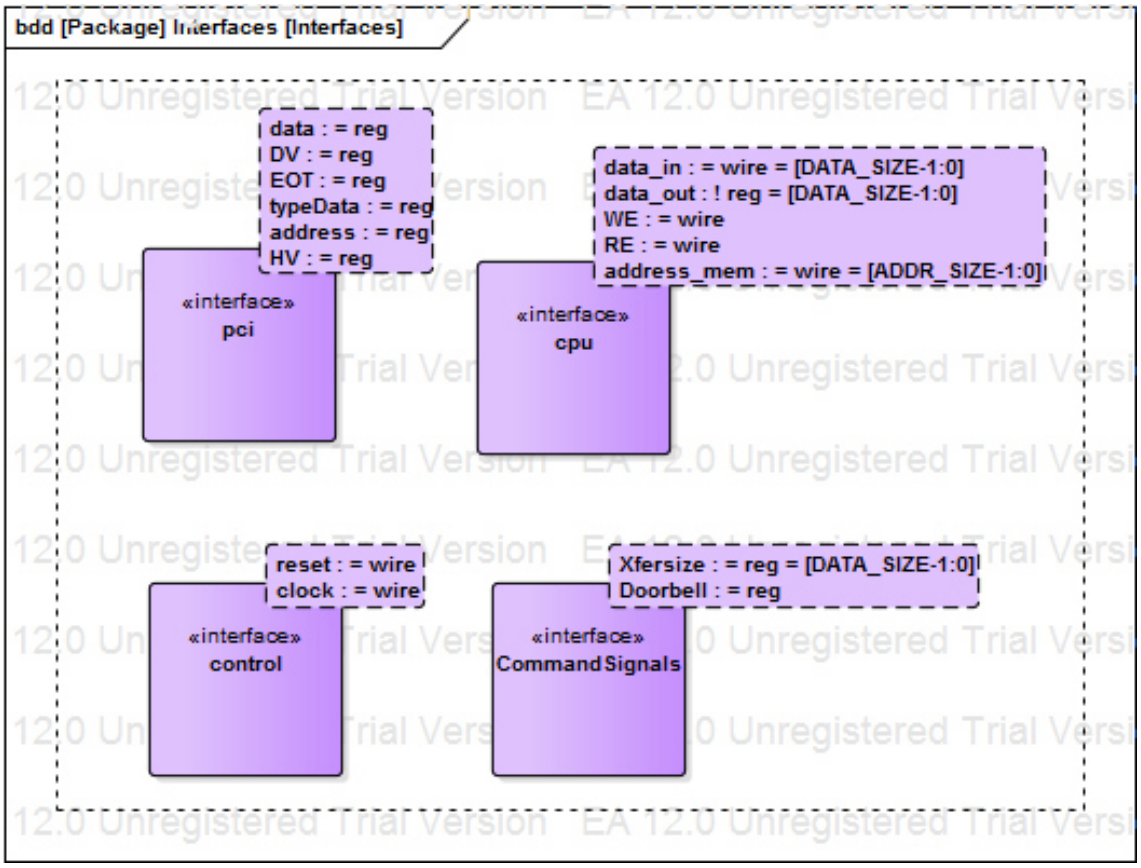

Figure 5.4: eDMA interfaces.

The *pci* interface is used to send read requests for the local memory. The Completions and Write requests uses two composite signals. These signals are composed by the six signals that comprise the *pci* interface. This approach can be an alternative to the interface mechanism, but it is not as clear as the use of interfaces since it is necessary to use an assign to merge these signals. The *cpu* interface is used for communication between the CPU and the eDMA, the *control* interface contains the *clock* and *reset* signals and, finally, the *CommandSignals* interface which is composed by the *Doorbell* and *Xfersize* signals used in all blocks of the first hierarchy level, that will be presented next.

The next step is to divide the solution into modules, since this example has not additional functionalities. This division is represented in figure 5.5.

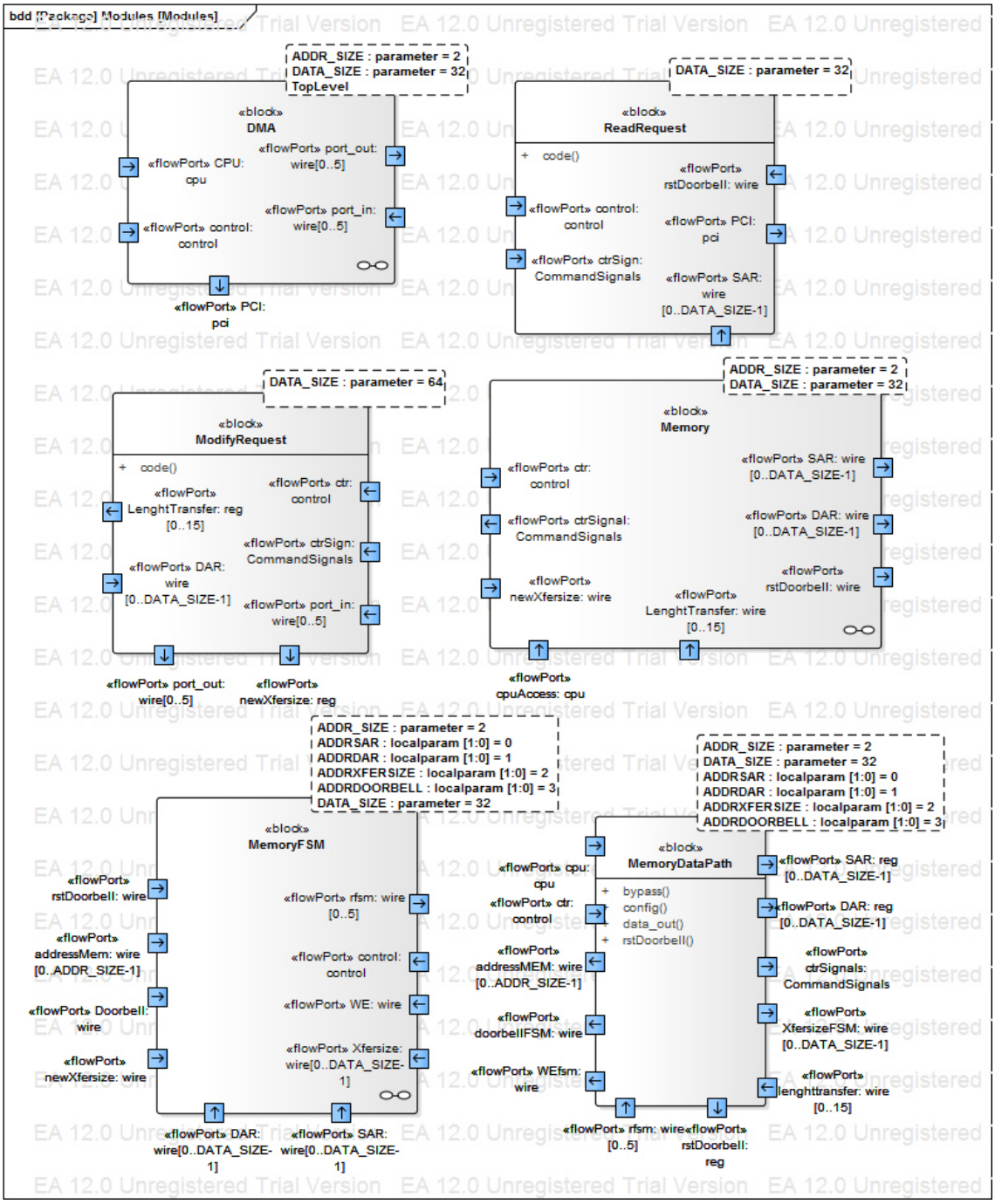

Figure 5.5: eDMA modules.

The top-level module has been divided into three main modules: a module with the configuration registers in the eDMA, the module that prepares the TLP's packets with read requests and it send them to the local memory and, finally, the module that receives the completion packets and transforms them into writing requests and sends them to the external memory. The figure represents 5.6 this first hierarchy level.

As can be analyzed by the figure, the application of the interface mechanism simplifies the

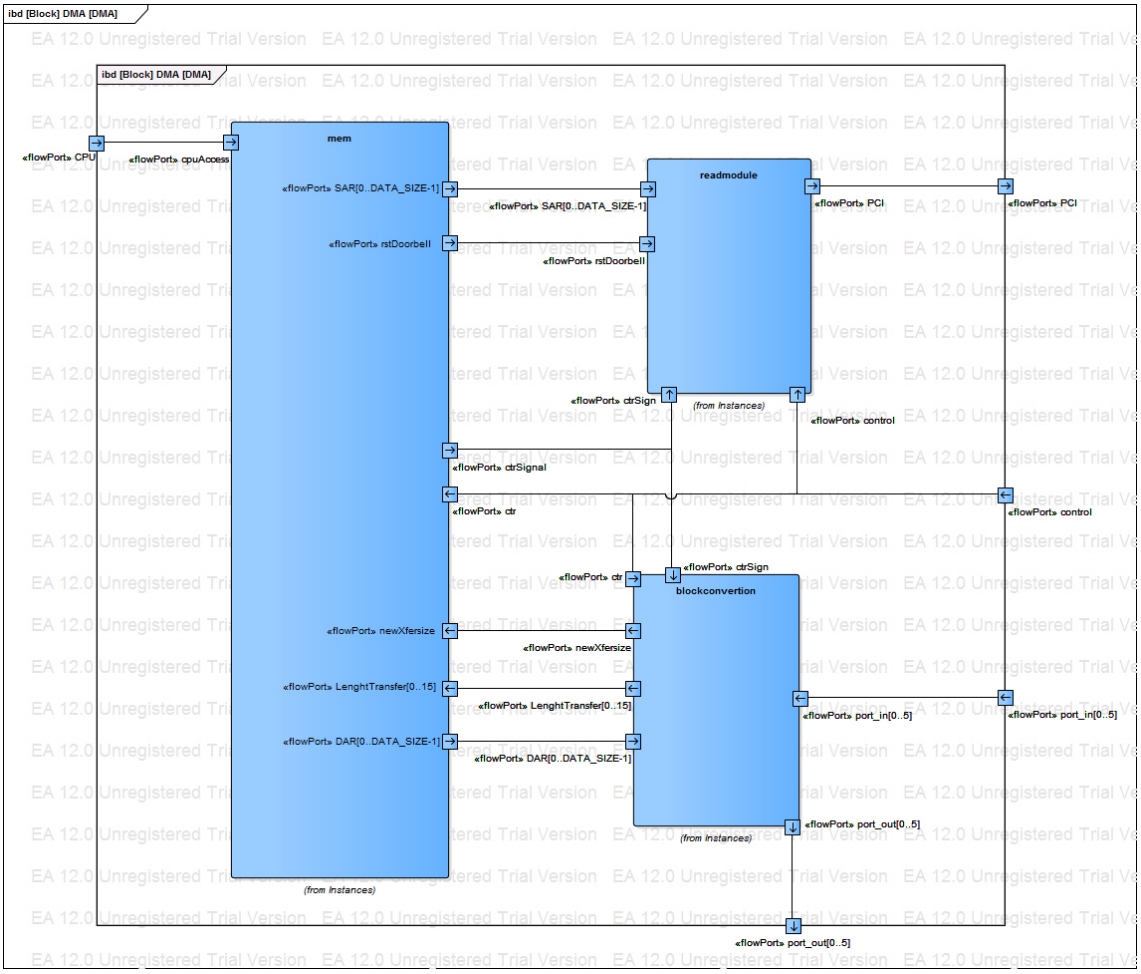

Figure 5.6: First hierarchical level of eDMA (Top-level).

representation of the interconnections between the modules which form the top-level module, becoming more clear the analysis of the diagram.

The Memory block is further divided into two other modules: a module with a finite state machine that controls the second module containing the data path of the memory. This second hierarchical level is represented in figure 5.7

The structure of the data path module is very similar to the modules presented in the example of the Adder. The big difference in this example is the presentation of the state machine. The figure 5.8 is the representation of the state machine that controls the memory module.

The FSM is responsible for generating control signals that will be used to adapt the data path to the desired function. The transition between states is triggered by the positive edge of the clock signal or the negative edge of the reset signal. This last signal takes precedence over the remaining signals. The operation performed, when this stimulus occurs, is defined by the designer in the signal properties. This operation is not shown in the diagram. Analyzing the diagram it is clear that state machine starts at the *waiting* state and it switches to the others states depending on the input signals. For example, the transition to the state *setSAR* occurs if *addressMEM* is equal to

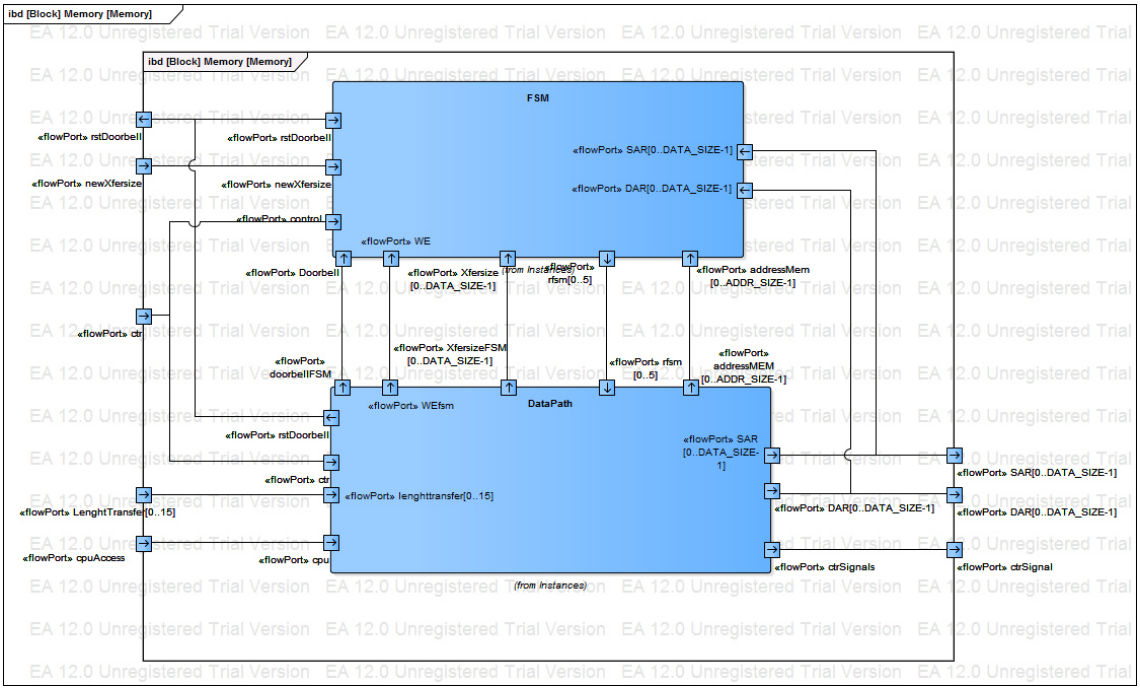

Figure 5.7: Second hierarchical level of eDMA (Memory).

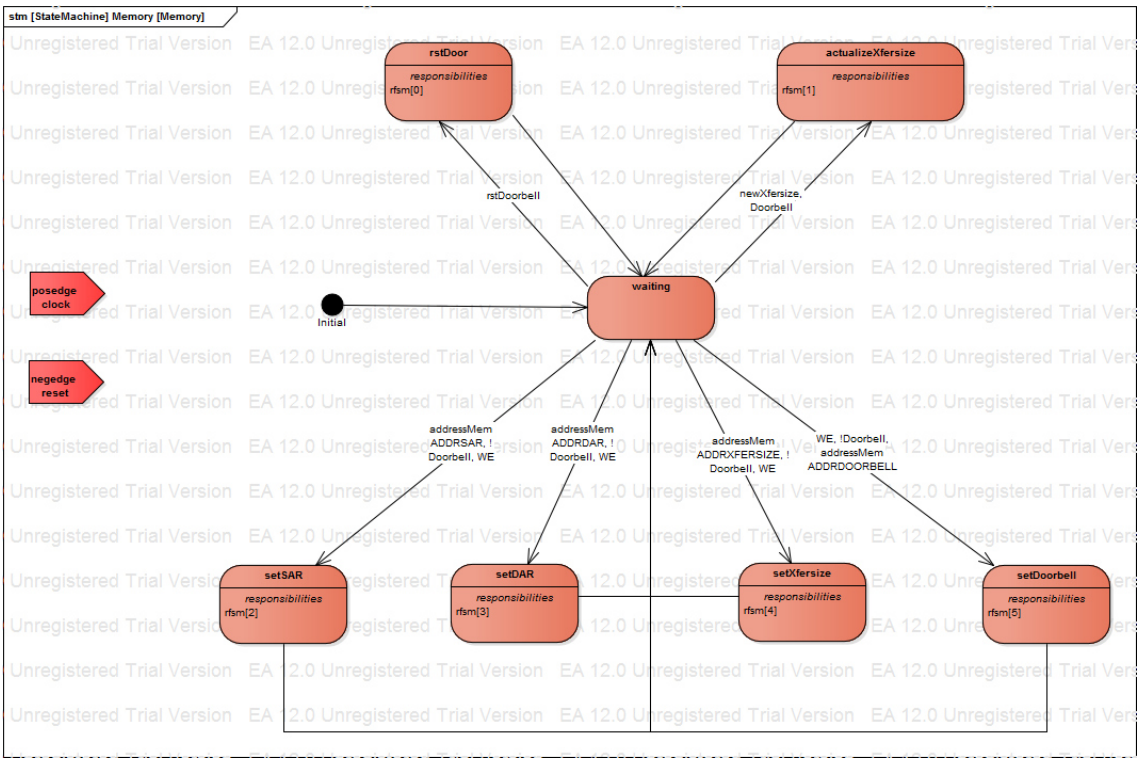

Figure 5.8: Finite-State Machine of the Memory.

ADDRSAR and *Doorbell* register is zero and the logic level of the *WE* signal is one.

If the designer wants to perform an alternative combination to make the same transition, it

would have to create a new connection between states. Each connection performs only the AND operations between signals, to perform the OR operations the designer must use two or more connections. For example, assuming that the transition $t1 = (AB) + (CD)$. The transition would be divided into two links, one with the condition *AB* and another with the condition *CD*.

The remaining transitions are similar to that described above. A note to the fact that be only use a single register to control the data path. As mentioned in the design rules of chapter 4, only the signals declared in the state take the value one, the rest take the value zero.

5.2.3 Code Generated

This example creates six Verilog files. Only will be analyzed two of the six files generated, since the others are similar to those analyzed in example of Adder.

The first code example is the top-level. In this example, it will be analyzed impact of the use of interfaces in SysML model.

```
module DMA #(
  //---- PARAMETER DECLARATION ----------------------------------
  parameter DATA_SIZE = 32 ,
  parameter ADDR_SIZE = 2
  )
  (//---- PORT DECLARATION ------------------------------------------
     input  wire  reset
    ,input  wire  WE
    ,input  wire  clock
    ,input  wire  [ADDR_SIZE-1:0]  address_mem
    ,input  wire  [DATA_SIZE-1:0]  data_in
    ,input  wire  RE
    ,input  wire  [5:0]  port_in
    ,output  wire  [5:0]  port_out
    ,output  reg  EOT
    ,output  reg  HV
    ,output  reg  address
    ,output  reg  DV
    ,output  reg  typeData
    ,output  reg  data
    ,output  reg  [DATA_SIZE-1:0]  data_out
  );

  wire w0_Doorbell;
  wire [15:0] w7;
  wire [DATA_SIZE-1:0] w0_Xfersize;
```

```
  wire w8;
  wire [DATA_SIZE-1:0] w6;
  wire [DATA_SIZE-1:0] w9;
  wire w10;

  ModifyRequest #(
.DATA_SIZE (32)
) blockconvertion (
  .reset(reset)
,.port_in(port_in)
,.clock(clock)
,.Doorbell(w0_Doorbell)
,.LenghtTransfer(w7)
,.Xfersize(w0_Xfersize)
,.newXfersize(w8)
,.DAR(w6)
,.port_out(port_out)
  );

  Memory mem (
  .reset(reset)
,.WE(WE)
,.clock(clock)
,.address_mem(address_mem)
,.Doorbell(w0_Doorbell)
,.data_in(data_in)
,.RE(RE)
,.LenghtTransfer(w7)
,.Xfersize(w0_Xfersize)
,.newXfersize(w8)
,.DAR(w6)
,.SAR(w9)
,.rstDoorbell(w10)
,.data_out(data_out)
  );

  ReadRequest readmodule (
  .reset(reset)
,.clock(clock)
,.Doorbell(w0_Doorbell)
```

```
,.HV(HV)
,.typeData(typeData)
,.Xfersize(w0_Xfersize)
,.EOT(EOT)
,.DV(DV)
,.SAR(w9)
,.rstDoorbell(w10)
,.data(data)
,.address(address)
  );

endmodule
```

Analyzing the code is perceived that all interfaces were replaced by their signals. This results in a increasing from five to fifteen ports.

Another important aspect to be noted is the impact of the interfaces on the declaration of the wires, as in the case of *CommandSignals* interface. In this case, as the interface is internal, that is, the interface is not connected to the external module, this receives a generic name. However, the interface is a set of signals. Each of these signals must be connected by different wires to not create short between signals. To prevent the appearance of this error, the wire name is completed with the signal name. This guarantees a clear connection.

Another important aspect to explain is the process of assigning the wires names. The wire names are created automatically with a numeric suffix appended to *w_*, that is generated globally for the whole design and independently of the module the wire belongs to. In this example, the module DMA has been created with wires named *w_0* and *w_6*, while all the wires generated with indexes 1 to 5 (from *w_1* to *w_5*) have been assigned to the module Memory (appendix D). Although this is functionally correct and will not compromise the rest of the design flow, the AutoGen tool will be improved in order to use net names explicitly assigned by the user in the SysML diagrams or, for unnamed nets, construct automatic names that include the identification of the modules connected by that wire.

Now it will be analyzed the code generated by AutoGen to encode FSM.

```
module MemoryFSM #(
  //---- PARAMETER DECLARATION ----------------------------------
  parameter DATA_SIZE = 32 ,
  parameter ADDR_SIZE = 2
  )
  (//---- PORT DECLARATION ----------------------------------------
     input  wire  reset
    ,input  wire  WE
    ,input  wire  clock
```

```
  ,input  wire  Doorbell
  ,input  wire  [ADDR_SIZE-1:0]  addressMem
  ,input  wire  [DATA_SIZE-1:0]  Xfersize
  ,input  wire  newXfersize
  ,input  wire  [DATA_SIZE-1:0]  DAR
  ,input  wire  [DATA_SIZE-1:0]  SAR
  ,input  wire  rstDoorbell
  ,output  wire  [5:0]  rfsm
);

localparam [1:0] ADDRDAR = 1;
localparam [1:0] ADDRDOORBELL = 3;
localparam [1:0] ADDRSAR = 0;
localparam [1:0] ADDRXFERSIZE = 2;

localparam [3:0] rstDoor = 0;
localparam [3:0] setSAR = 1;
localparam [3:0] actualizeXfersize = 2;
localparam [3:0] setDoorbell = 3;
localparam [3:0] setDAR = 4;
localparam [3:0] setXfersize = 5;
localparam [3:0] waiting = 6;
reg [3:0] state;
reg [3:0] nextState;

always @( negedge reset or posedge clock )
begin
  if( !reset )
  begin
    state <= waiting;
  end
  else
  begin
    state <= nextState;
  end
end

always @*
begin
  case( state )
```

```
      rstDoor:
        nextState <= waiting;
      setSAR:
        nextState <= waiting;
      actualizeXfersize:
        nextState <= waiting;
      setDoorbell:
        nextState <= waiting;
      setDAR:
        nextState <= waiting;
      setXfersize:
        nextState <= waiting;
      waiting:
      begin
        if( rstDoorbell )
          nextState <= rstDoor;
        else  if( Doorbell  & newXfersize )
          nextState <= actualizeXfersize;
        else  if( !Doorbell  & addressMem == ADDRDOORBELL & WE )
          nextState <= setDoorbell;
        else  if( !Doorbell  & addressMem == ADDRDAR & WE )
          nextState <= setDAR;
        else  if( addressMem == ADDRXFERSIZE & WE  & !Doorbell )
          nextState <= setXfersize;
        else  if( addressMem == ADDRSAR & !Doorbell  & WE )
          nextState <= setSAR;
      end
     default: nextState <= waiting;
    endcase
  end

  assign rfsm[0] = (state == rstDoor);
  assign rfsm[1] = (state == actualizeXfersize);
  assign rfsm[4] = (state == setXfersize);
  assign rfsm[5] = (state == setDoorbell);
  assign rfsm[3] = (state == setDAR);
  assign rfsm[2] = (state == setSAR);

endmodule
```

The code generated by the AutoGen can be divided into 4 blocks. The first is the definition

of all parameters necessary for the proper functioning of the FSM. The first parameters declared are the addresses of the registers set by the designer. Then the parameters related to codification of the states and the registers that keep the information about the current state and the next state are declared. The following block defines the next state. Code associated to the trigger signals are defined in this block, in this case the reset signal. The third block is responsible by calculating the next state depending on the transition conditions. And finally, the last block sets the outputs in each state of the FSM. This organization aims to maintain the readability and clarity of the code.

This is a common organization to all state machines generated by the Autogen. If the outputs are not defined in the diagram, as in the case of Mealy's machines, the last code block is generated manually by the designer.

5.3 Conclusion

The two examples given help to understand the application of the rules set out in the previous chapters. The first example summarizes the application of the techniques for reconfiguration of IPs using the compiler directives, the introduction of Verilog code in model and the construction and generation of a hierarchy Verilog modules from SysML model. The second example summarizes the interfaces application and illustrates the advantage on using them. The later example also shows the application of the State Charts diagram for encoding FSM. With these two examples, it is possible to understand some of the strengths of AutoGen tool, along with SysML modeling language, simplifying the development of SoC subsystems.

Chapter 6

Final Conclusions

6.1 Analysis of the Developed Work

The main objective was to create an automatic generation tool of Verilog code based on SysML diagrams. This tool automates the generation of hardware description code, which leads to reducing the development time of a product. On the other hand, the creation of a module in SysML of one system is time consuming, but this allows the tool to generate the code. In fact, creating a graphical model of a project is something quite usual for a designer. Typically, before starting to describe a hardware, a designer does a small drawing of the system structure. Basically, the main idea of this work is to encourage designers to make these drawings in SysML with a little more accuracy, motivated by the fact that later they can use these same drawings to generate the HDL code.

These diagrams can also be used to document the system structure and some of them can even be used to document their behavior, as in the case of the State Charts diagrams. Some diagrams can even be used in meetings with stakeholders, so as to favor mutual understanding between designers and stakeholders.

Sharing the time of documentation and development leads to reducing the project time, due to this dual use of the diagrams. At the same time, the perception of documentation increases because it becomes more graphical and less textual.

Using both SysML and AutoGen provides the possibility to doing documentation and development both at once, instead of worrying about each one of them individually. In most cases, the initial specifications evolves with the project. So using this approach, the documentation and the code are always synchronized on every iteration (product release cycle, or requirements change from the stakeholders). This also helps improving the quality of the product and leads to a benefic reduction of the time-to-market.

Table 6.1 illustrates the comparison between the traditional design flow and this new flow which uses the SysML language along with the Autogen.

The code generated from the SysML model guarantees that the RTL model corresponds entirely to the SysML model. However, during development, the RTL code can be modified by

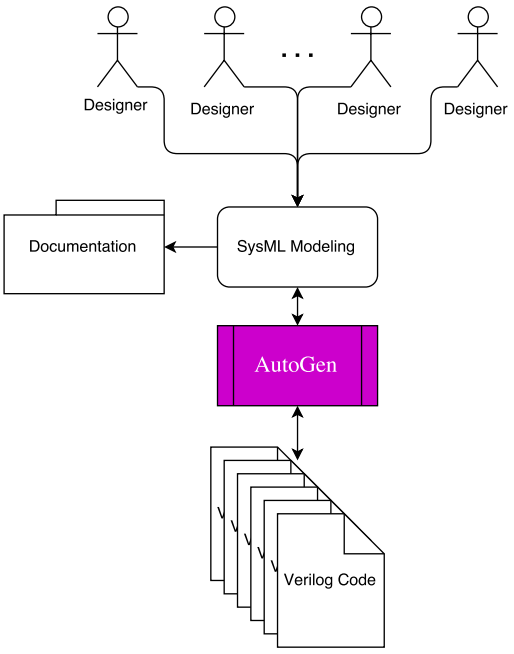

Figure 6.1: Design flow driven by AutoGen tool.

designer, for example, fix a bug or correct a minor functional error. This can lead to inconsistency between the models if the change is not also carried out in SysML model. This inconsistency can be detected using the verification functionality built into the tool. This way, the designer can always check if the two models, RTL Verilog and SysML, are in agreement.

The AutoGen will simplify and improve the performance of some critical steps of Synopsys design flow, as shown in figure 6.2.

The major difficulty encountered throughout this work was finding simple but efficient mechanisms not too constrained by the SysML and EA modeling rules. It is hard working with the files generated by the Enterprise Architect because this tool has not been designed for the same purpose of this work and thus it has been necessary to adapt some of the constructs to fulfill the needs for RTL generation and handling the compiler directives.

6.2 Future Work

For future work, in terms of the FSM generation functionality, the State Charts diagrams could be better exploited in order to simplify the rules of drawings for this type of diagram. At the same time, other mechanisms could be created to generate other types of machines beyond Moore, such as Mealy machines or hierarchical FSMs. This feature could also be improved in order to support sequential machines or finite state machines.

Some aspects of the generation of hierarchical modules can be improved, such as the naming of the wires and the algorithm that identifies connections that represent the same point in RTL model.

Table 6.1: Comparison between the traditional model and the new design model.

	Traditional Flow	SysML with AutoGen
Focus	Code	Documentation
Connection errors	Frequently	Does not exist
Encoding FSM	Difficult	Easy
Creation of hierarchical models	Difficult	Easy
Interconnection between instances	Difficult	Easy

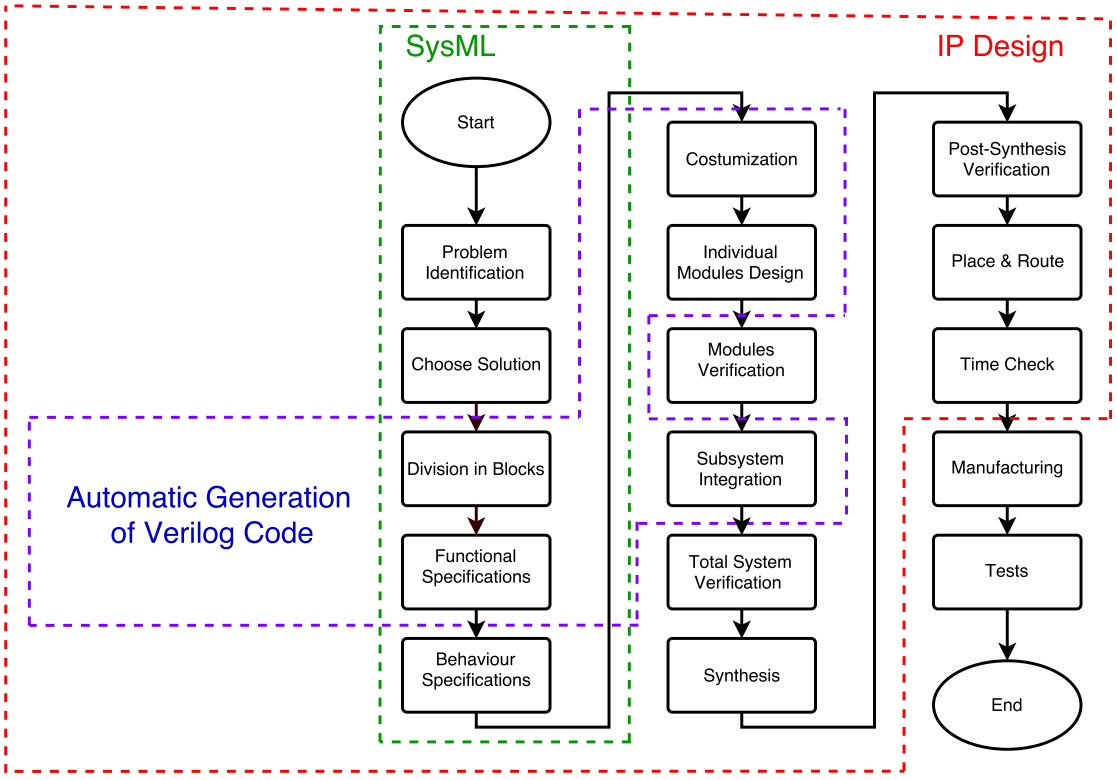

Figure 6.2: AutoGen impact in Synopsys design flow.

However, the main thing would be to create an unique tool in which it would be possible to draw the SysML diagrams and, at the same time, generate Verilog code. Thus, it could be possible to create mechanisms of drawing, which would improve the quality of the documentation based on SysML diagrams and, at same time, would simplify the generation of Verilog code, thus, the development team would use a unique tool to support the development of systems or subsystems on-chip.

Other SysML diagrams, such as the sequence or activity diagrams, could also be used to automatically generate basic verification stimulus. (Testbench automation)

Appendix A

Verilog Code of Adder

```
`define C_IN
`define OVL
`define C_OUT

module adder #(
  //---- PARAMETER DECLARATION ----------------------------------
  parameter N = 4
  )
  (//---- PORT DECLARATION ----------------------------------------
     input   wire  [N-1:0]  A
    ,input   wire  [N-1:0]  B
    ,output  wire  [N-1:0]  Sum
  `ifdef C_OUT
    ,output  wire  Cout
  `endif
  `ifdef C_IN
    ,input   wire  Cin
  `endif
  `ifdef OVL
    ,output  wire  ovl
  `endif
  );

 `ifdef C_OUT
   `ifdef C_IN
      assign {Cout , Sum} = A + B + Cin;
   `else
  assign {Cout , Sum} = A + B;
```

```
    `endif
  `else
    `ifdef C_IN
   assign Sum = A + B + Cin;
    `else
   assign Sum = A + B;
    `endif
  `endif

  `ifdef OVL
assign ovl = ( A[N-1] == B[N-1] ) && ( A[N-1] != Sum[N-1] );
  `endif

endmodule
```

Appendix B

Model in SysML

This chapter aims to show an example of how the SysML can be used to document a design SoC. The development of this modeling follows the model suggested by ICONIX at document [1]. This model, as shown in appendix C.1, begins with the modeling of requirements, followed by behavior and structure and, finally, restrictions.

B.1 Requirements Model

The requirements model aims to structure the requirements. This diagram allows an easy communication between the designer and the stakeholders in order to refine requirements of the problem and ensure that the designer understands what stakeholders want with the project. The sequence of steps for modeling the requisites is described at the figure in appendix C.2.

The requirements diagram of the DMA is shown in figure B.1.

This type of diagram allows refine the specifications as much as you wish. There are several types of connections that can be used to improve details in diagram. This diagram represents details about physical aspects, such as interfacing with the memories and the name of registers, and functional aspects, such as ease of configuration or the creation and modification of TLP packets. The requirements may also be connected to other diagrams, such as block diagram. With these connections the designer can create a matrix that show if a requirement was implemented in final product, using the Enterprise Architect.

One example of specifications related to the memory block in the diagram of figure B.1 (the grey box in the left side) is to have a simple interface to allow configure the DMA. This specification is observed in diagram as a functional requirement and it is partitioned in three generic ports: *Enable Operation*, *Address Memory Line* and *Data line*. On its turn, the *Enable Operation* is divided in *Write Enable* and *Read Enable*. Each one extends the physical requirement *Enable Operation*. Finally, it is clear to infer that the *Memory Block* contributes to satisfy the main requirement *Configuration*.

The next step in our project is creating the behavior model.

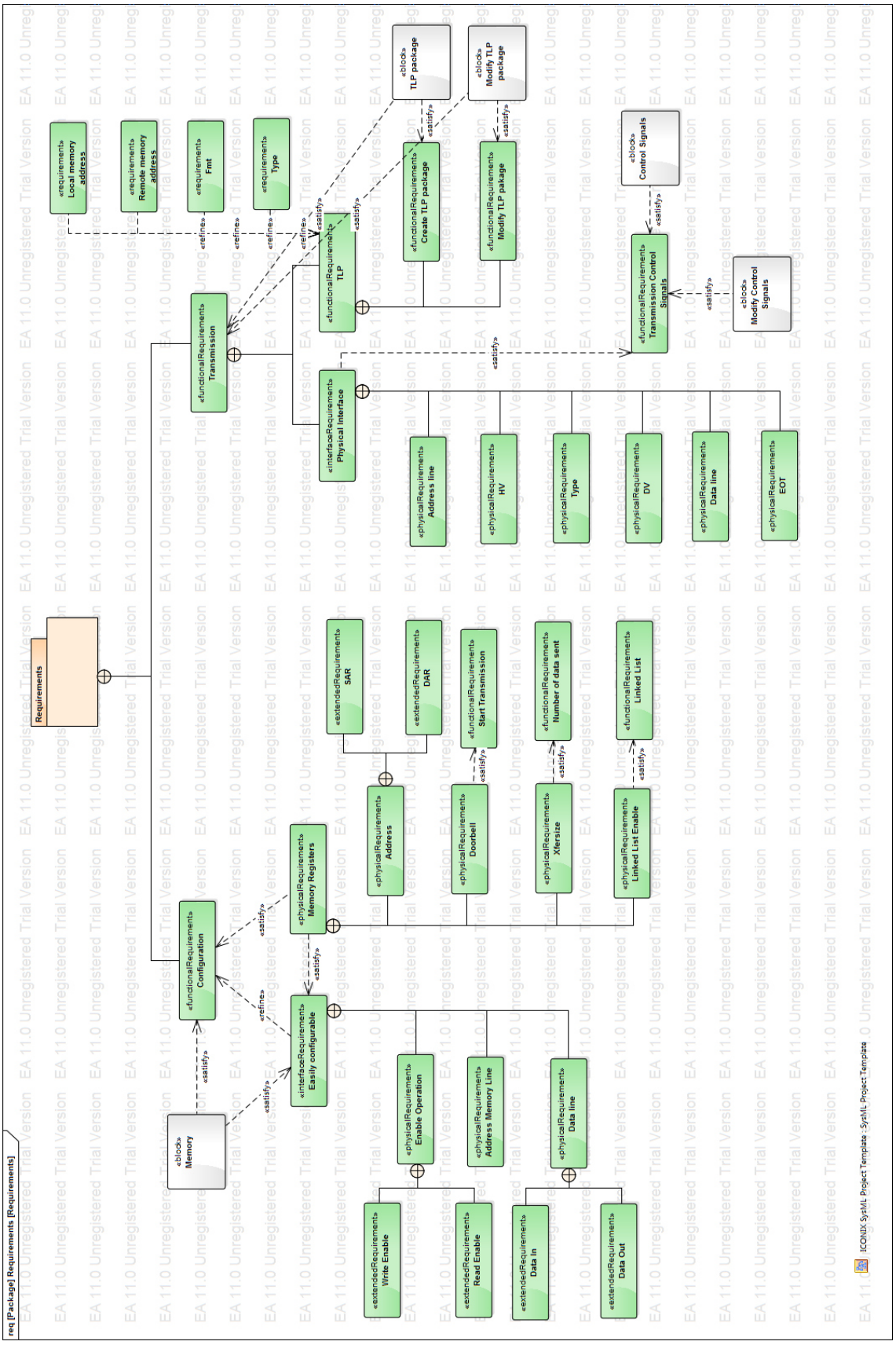

Figure B.1: Requirements Diagram

B.2 Behavioral Model

In this set of diagrams, the designer can model the behavior that the system will have after its creation. The figure in appendix C.3 indicates the steps that was taken to carry out the modeling.

B.2.1 Use Case Diagram

Starting at the left branch, the first diagram designed is the Use Cases Diagram.

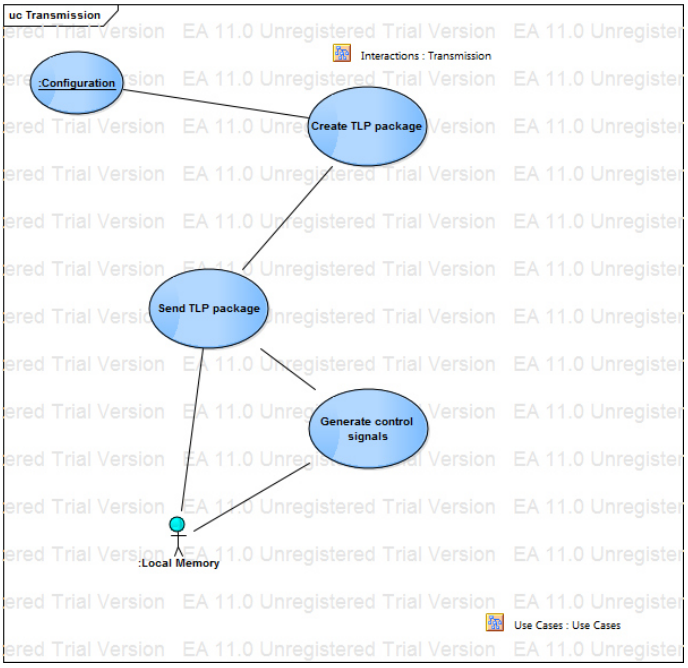

Figure B.2: Use Case Diagram

In this diagram, it is possible to show the interaction between different entities of the system and the behavior of this, without express the order of sequences.

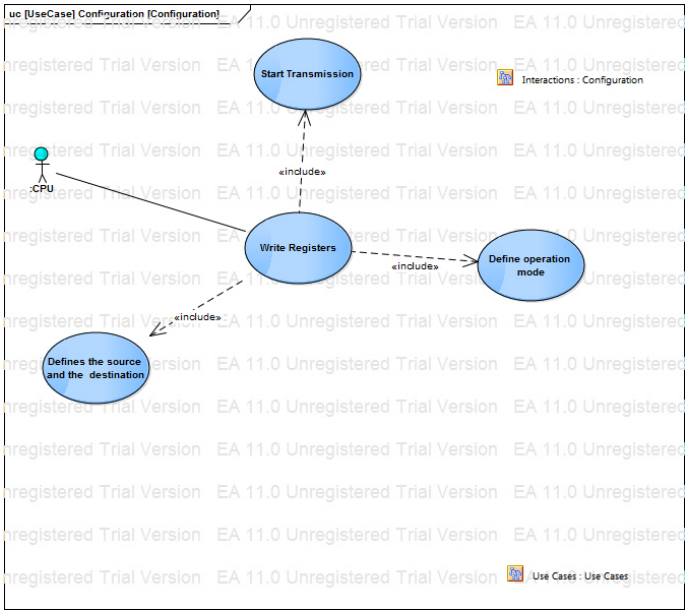

Figure B.3: Configuration Use Case

Analyzing the previous diagram, it is easily understood that the configuration is an interaction between the CPU and system memory. Configuration consists of writing data into memory locations. With this action it is possible configure the local and external memory address, the operating mode and start transmission.

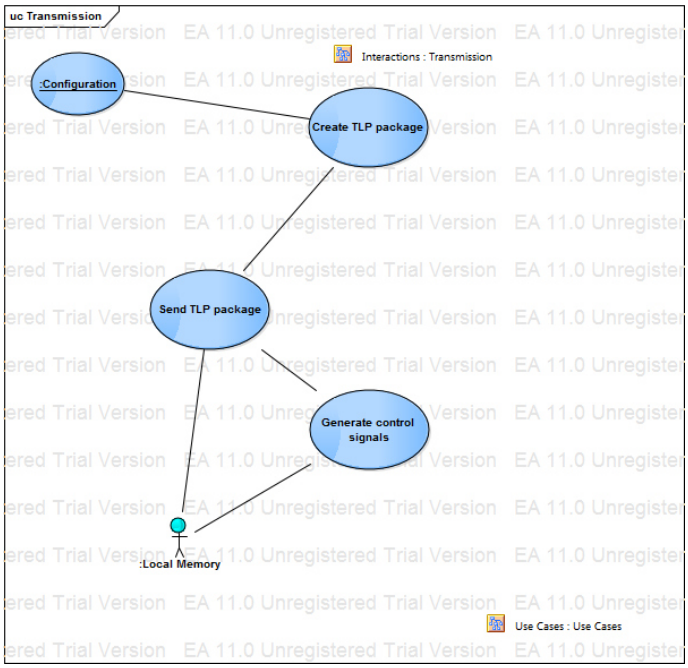

Figure B.4: Transmission Use Case

With this use case it is possible see that the transmission is initiated by a system entity - configuration. This action consist in create a TLP packet and his send. With packet is sent control

signals to local memory.

As can be seen in figure B.5, both signals will be change and send to external memory.

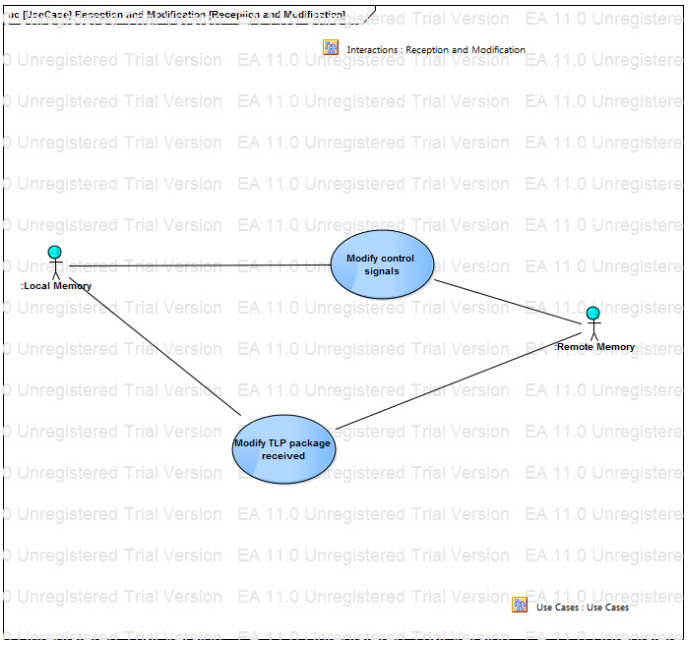

Figure B.5: Modification Request Use Case

B.2.2 Interaction Diagrams

This type of diagrams aims to describe the order of the interactions in the system.

The figure B.6 shows the necessary interactions between the CPU and the system for DMA configuration.

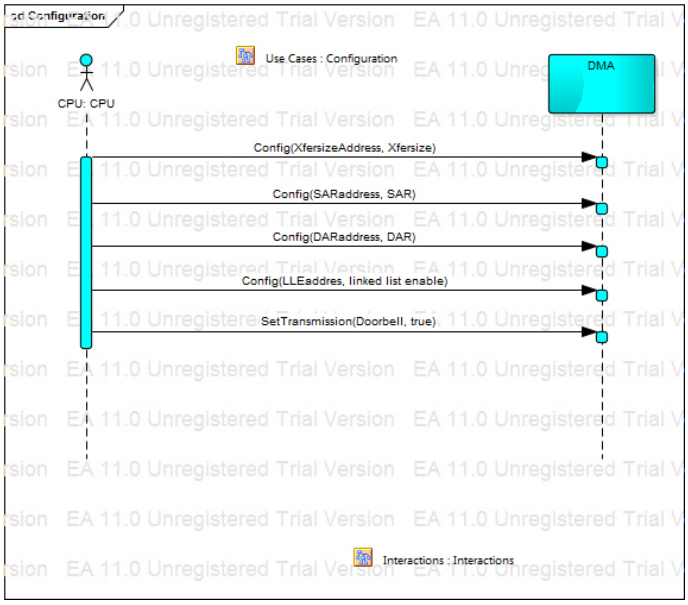

Figure B.6: Configuration Interaction

The figure B.7 shows the interactions between the system and local memory after the beginning of transmission.

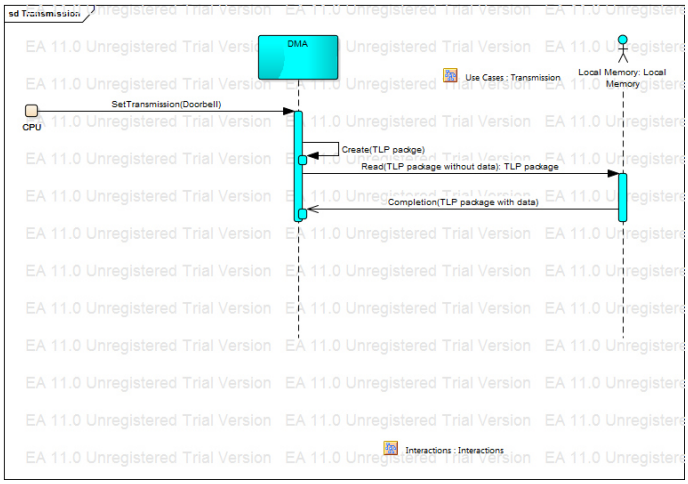

Figure B.7: Read Request Interaction

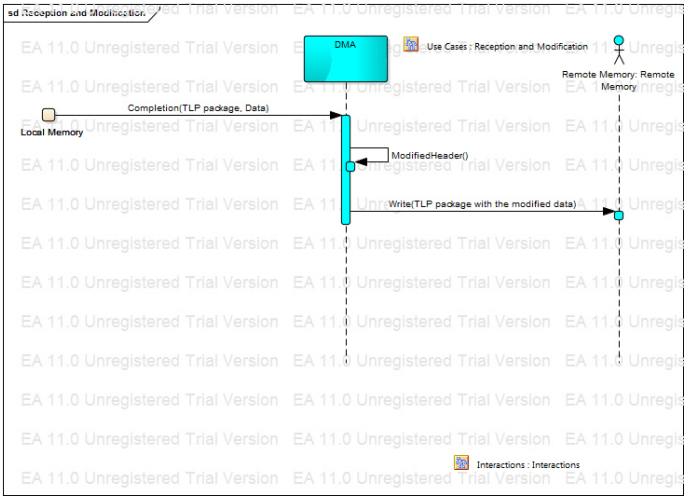

Figure B.8: Modify Request Interaction

The last diagram, B.8 allows to see the interaction that exists between packet sent by local memory and packets sent by DMA.

B.2.3 State Machine Diagram

The next diagrams seek to describe the sequence of system states.

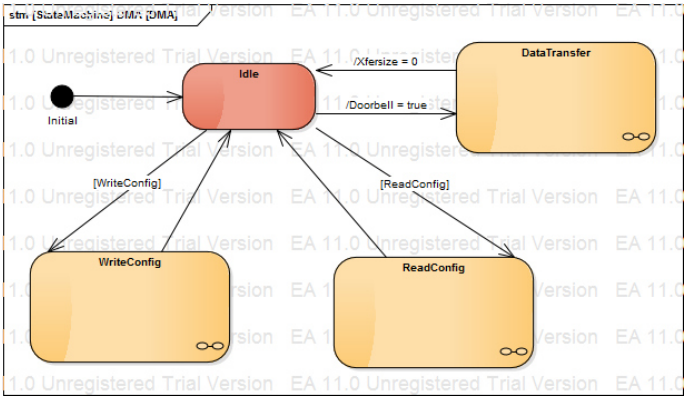

Figure B.9: State Machine

The external stimuli are defined by square brackets while the internal stimuli are defined by a slash followed by the name. Looking at the diagram, it is possible see that the read or write operations are initiated by external stimuli while the transfer operation is initiated by the *Doorbell*.

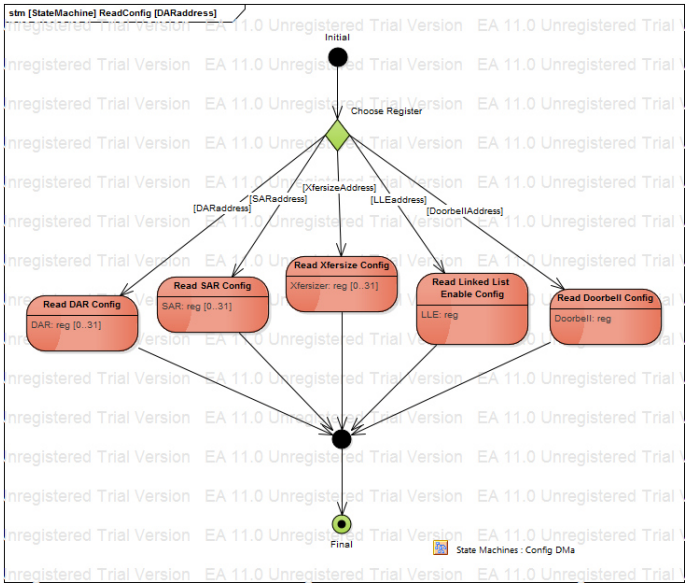

Figure B.10: Read Configuration State Machine

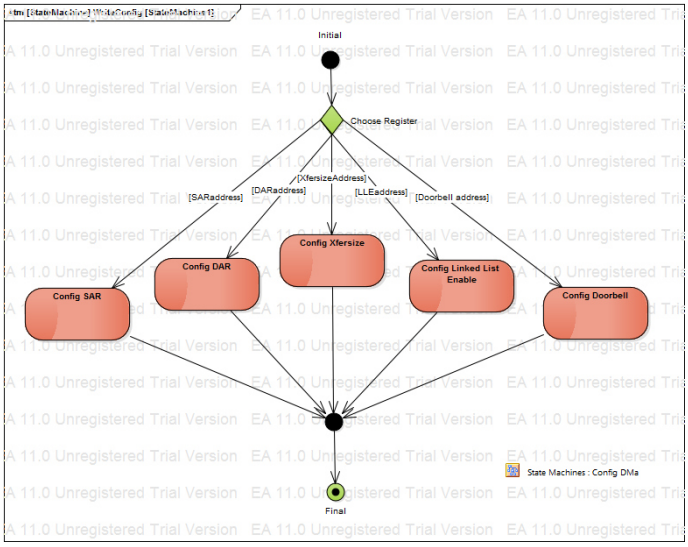

Figure B.11: Write Configuration State Machine

The first diagram above represents the system configuration and, the second, represents reading settings.

The following diagram describes the data transfer in system. The transfer mode of the system is defined by *LLE* flag (enable linked list mode).

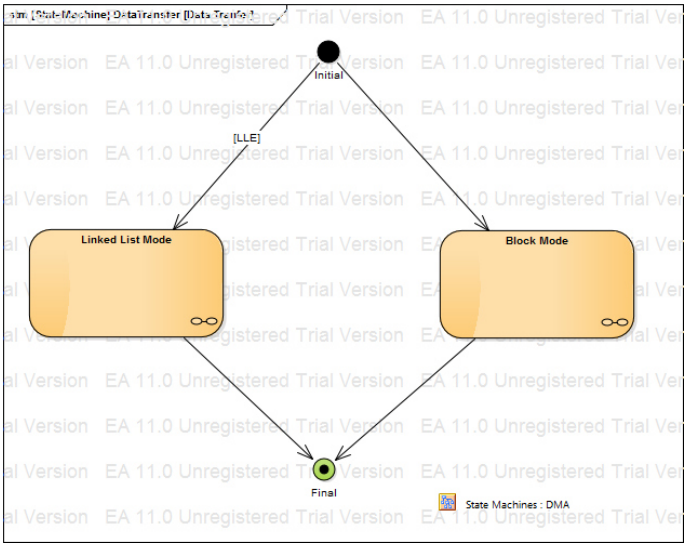

Figure B.12: Data Transfer State Machine

If *LLE* flag was activate, the process starts transfer on linked list mode. After read the element, if the *LLP* flag (linked list pointer) is active or the *CB* flag is equal to *CCS* the type of element is verified, if not the process ends. In the case of a pointer element, the local memory address is updated and the process is restarted, otherwise the process continues on block transfer mode. At the end of the operation, the pointer is updated. The following figure tries to show this process.

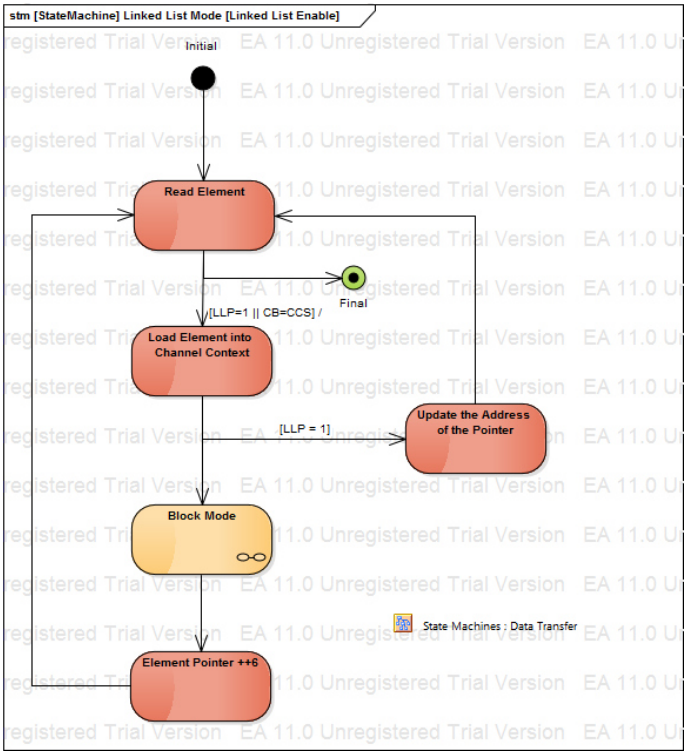

Figure B.13: Linked List Mode State Machine

In the case of block transfer mode, the system begins by creating the package to send. Then the system sends the package along with other signals. After sending all signals, the system stays awaiting a response from local memory. After receive the answer, the DMA starts to modify and send the signals received. The following figure tries to show that.

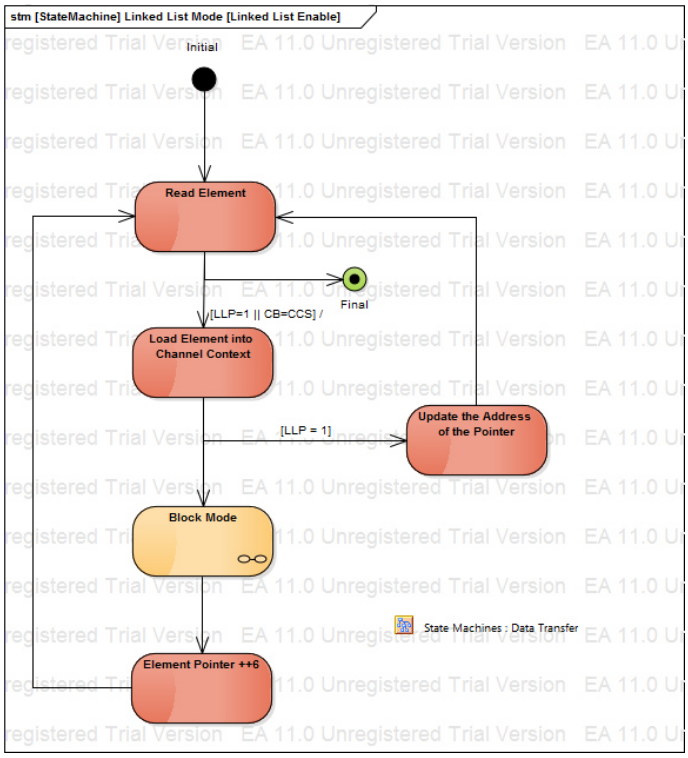

Figure B.14: Block Mode State Machine

B.3 Structure Model

After completed the analysis of system behavior, it is time to think how many blocks are needed to implement the system and its function.

For this, it will be followed the steps described in appendix C.4 to modeling the system structure.

B.3.1 Problem Domain

Before beginning the division into blocks, it is necessary a reflection on the problem domain. For this, it is created a model to represent the problem domain.

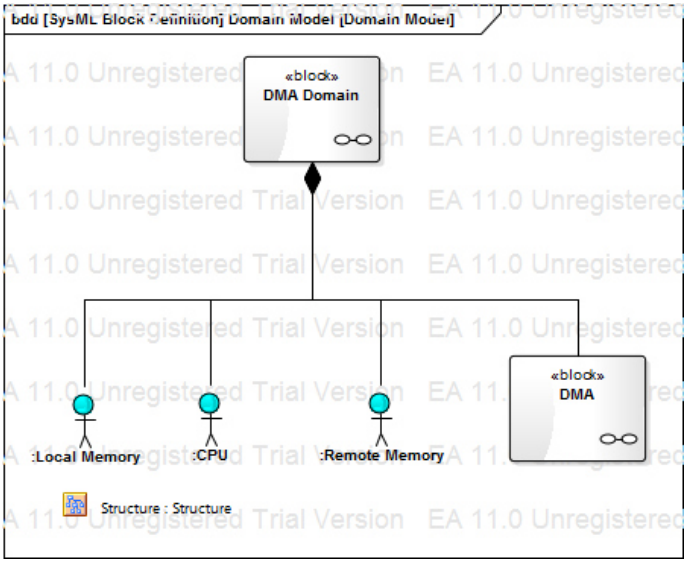

Figure B.15: Problem Domain

For this problem, the diagram is not very complex but it is an aid in the visualization of elements that interact with the system to be developed.

If it was necessary, it can be created another diagram for understand the interfaces between the system and entities that interact with it.

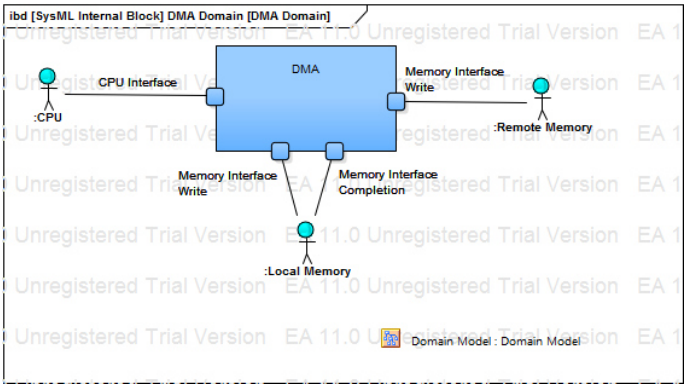

Figure B.16: DMA Domain

So, it is created an idea of how many interfaces are required and their connections.

B.3.2 Blocks Diagram

The following diagram illustrates the system structure through blocks. At this time, the blocks are still abstract entities which may not correspond to blocks that will be produced but are crucial and will be contained on final product.

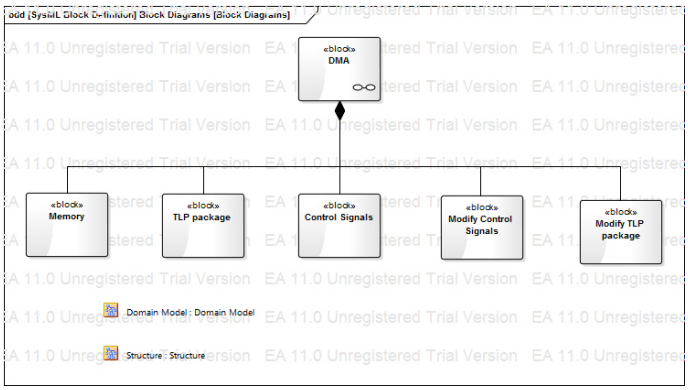

Figure B.17: Block Diagram

Now, it is already possible to develop a more detailed block model of the final product.

B.3.3 Internal Blocks Diagram

In this diagram, it is possible understand which modules are needed to develop and their interfaces, defining the direction and size of ports.

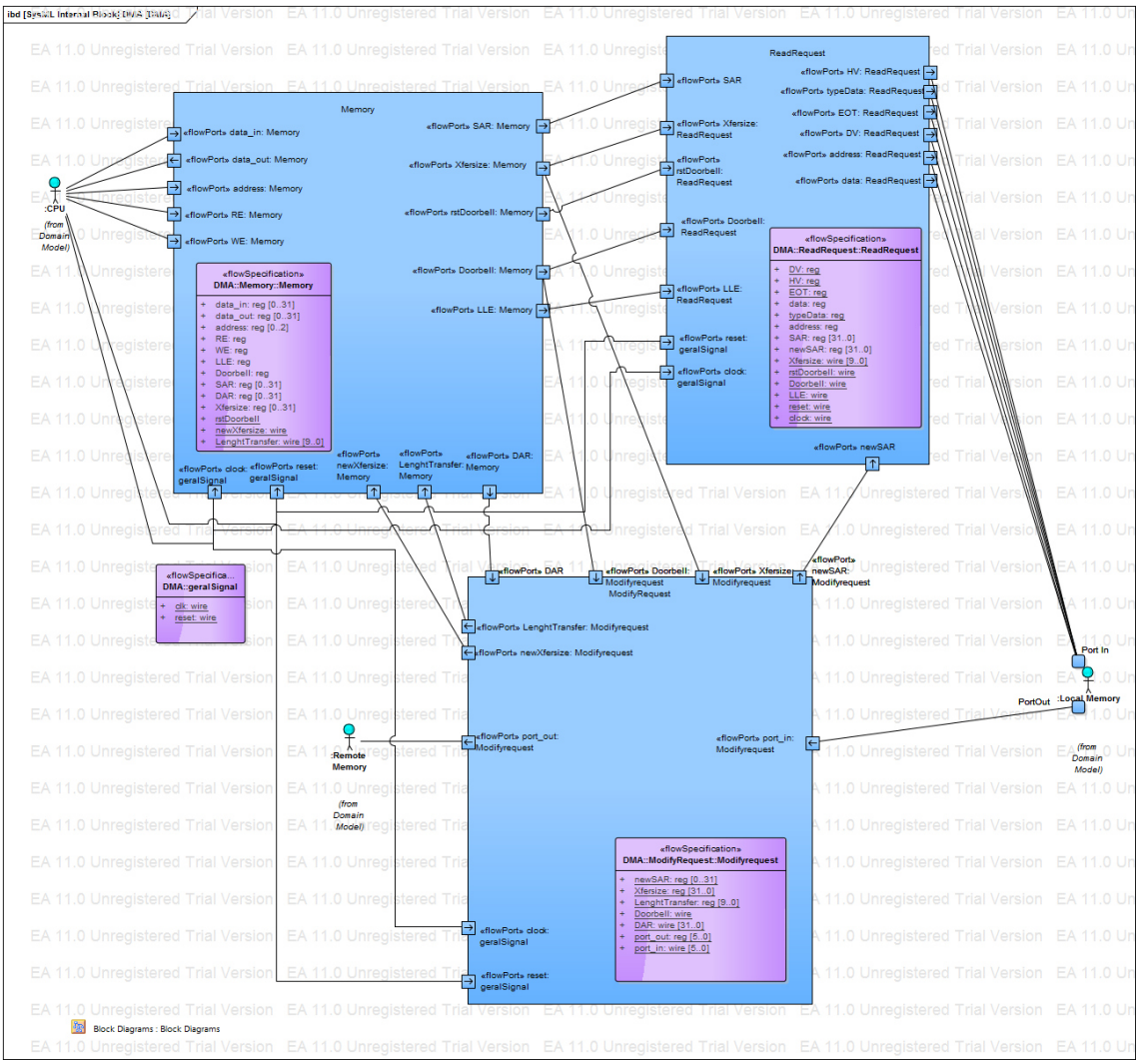

Figure B.18: Internal Blocks Diagram

B.4 Activity Diagram

By definition, this diagram belonging to the group that describes the system behavior. This diagram can be used as a bridge the gap between the behavior and structure diagrams. The importance of the diagram increases with the complexity of system activities. This contains information about blocks and their behavior, highlighting signals that initiate activities in the system. The following diagram is an example of this.

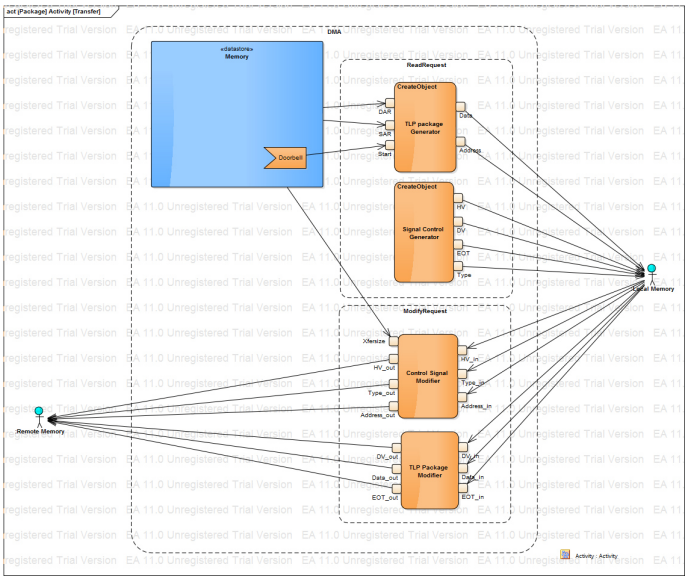

Figure B.19: Activity Diagram

B.5 Constraints and Parameters

Now, it is time to define the constraints and parameterization of the model. As suggested in appendix C.5, in this section, the designer can define the mathematical models of the system to create and perform simulations. This is a feature of the Enterprise Architect but for this problem, is not very useful since it is not able to work at the level of abstraction that system works. These simulations are useful to work with macro-model because, for example, it is possible multiply and manipulate sinusoidal functions. This tool is not ready to receive and perform operations with digital signals.

However, the user can define the instantiation module, that is to say, the module name and its interfaces, as shown in the following figure.

B.5.1 Constraints Blocks

This diagram is particularly useful to compare with generated module and verify that all interfaces are created.

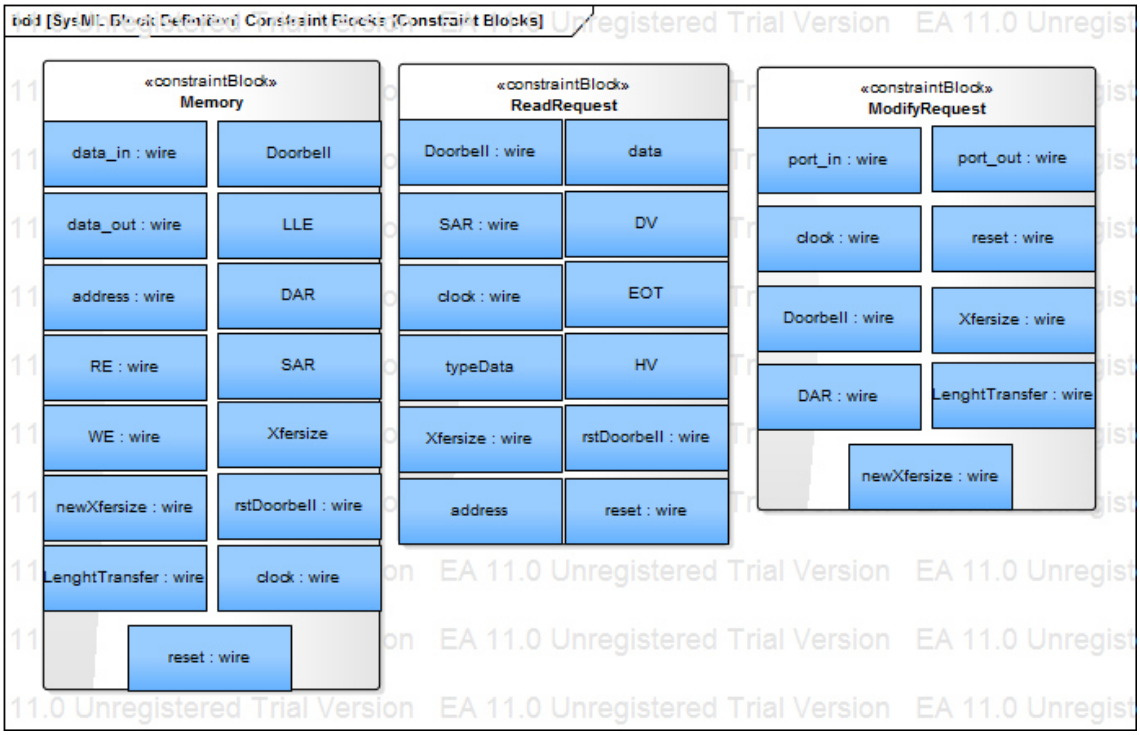

Figure B.20: Constraints Blocks

B.5.2 Parametric Diagram

It is now possible to connect generated blocks and to create a representation of final system, like figure B.21.

With this diagram it is possible to verify the ramifications of signals, which signals are internal and what are the signals that attach to the outside. With orientation of arrows is also possible to see the flow of information and thus understand which signals are input and output.

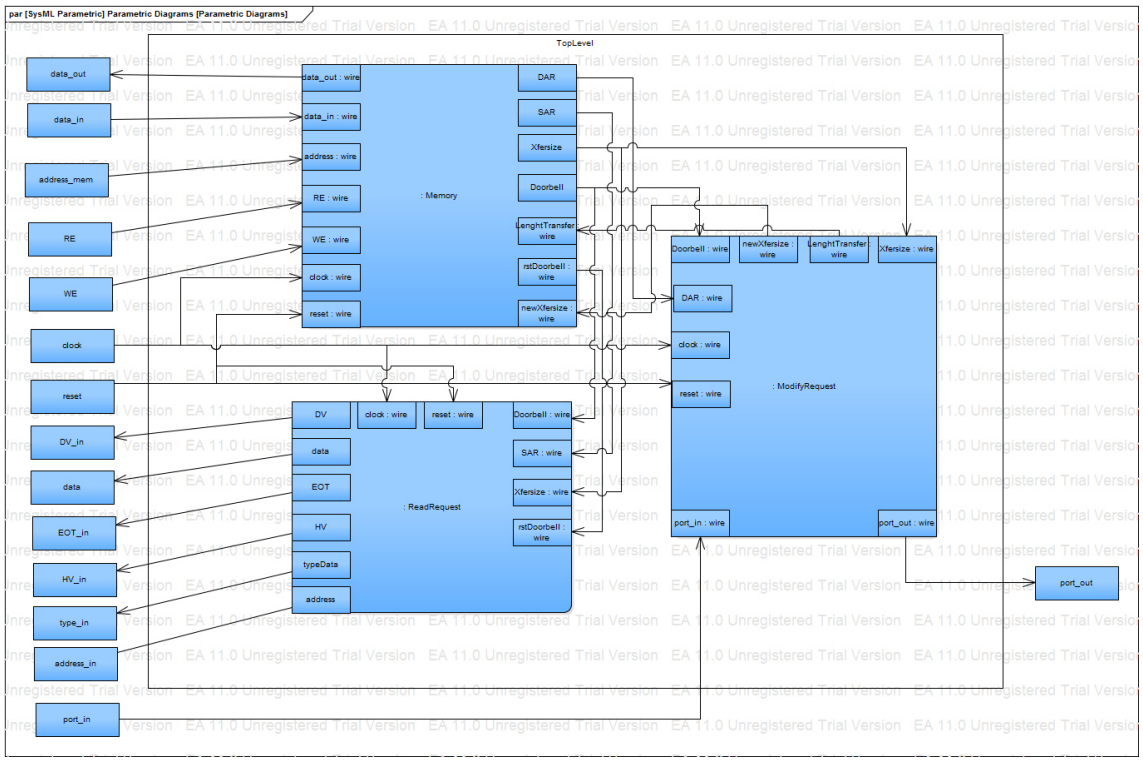

Figure B.21: Parametric Diagram

Appendix C

Embedded Systems Development

C.1 Roadmap for Embedded Systems Development

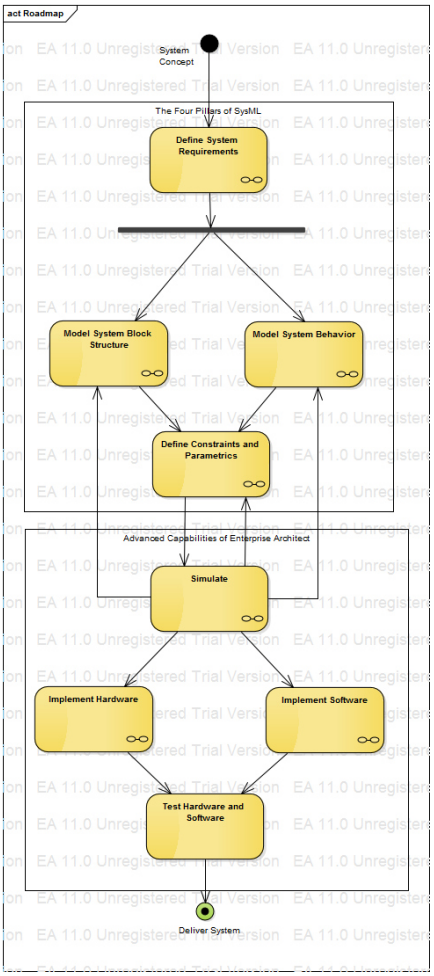

Figure C.1: Roadmap for Embedded Systems Development[1]

C.2 Requirements Model

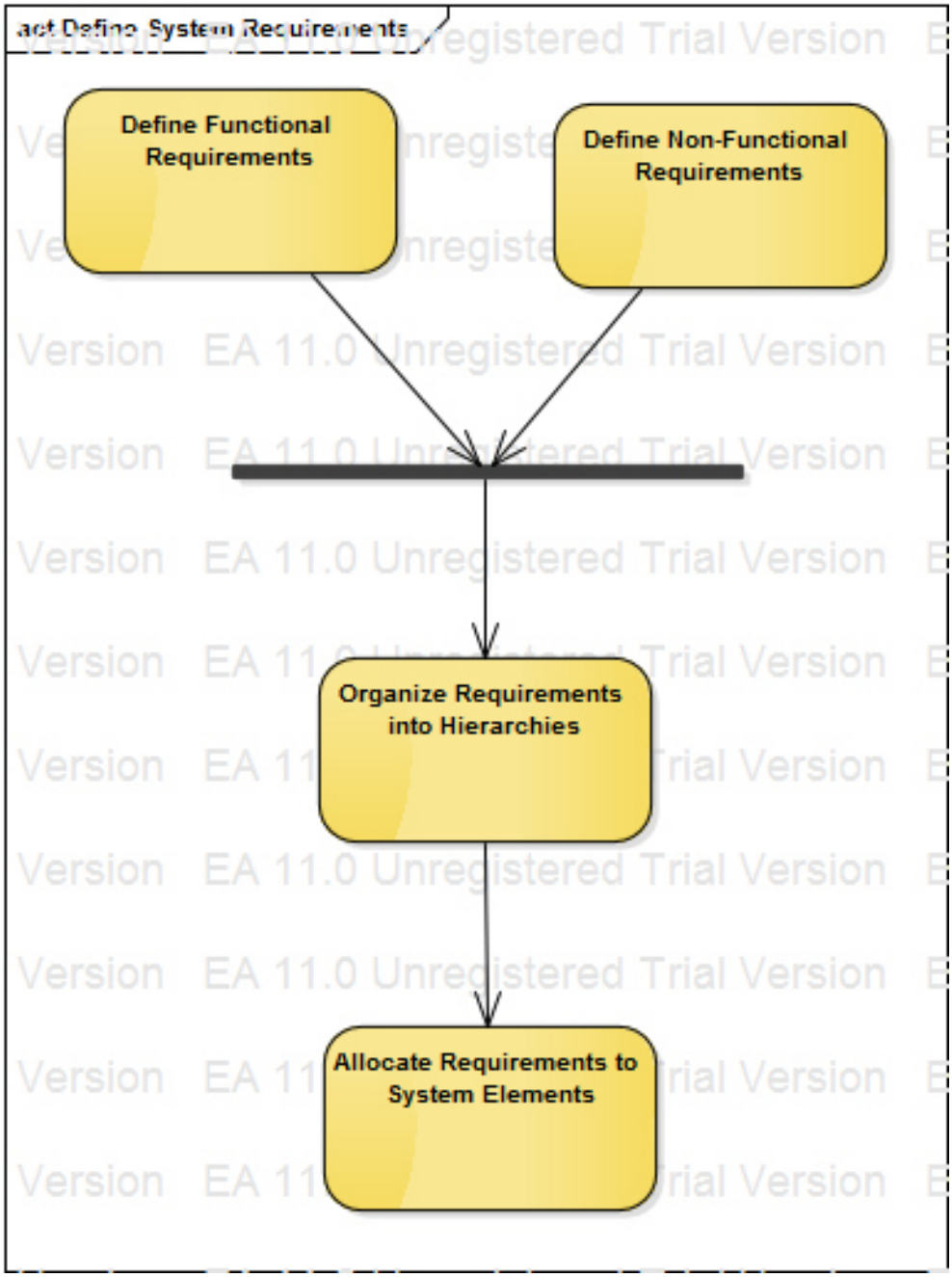

Figure C.2: Requirements Model[1]

C.3 Behavioural Model

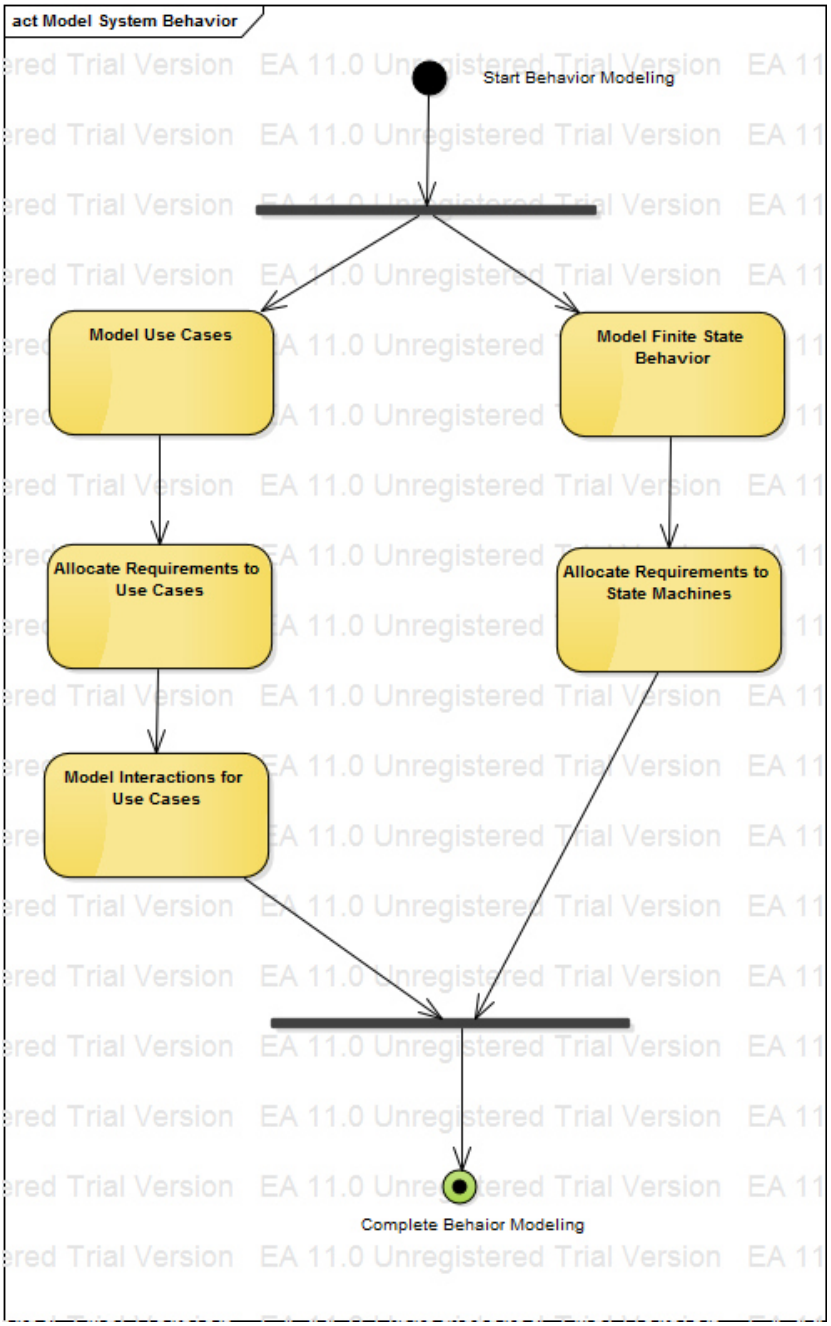

Figure C.3: Behavioural Model[1]

C.4 Structural Model

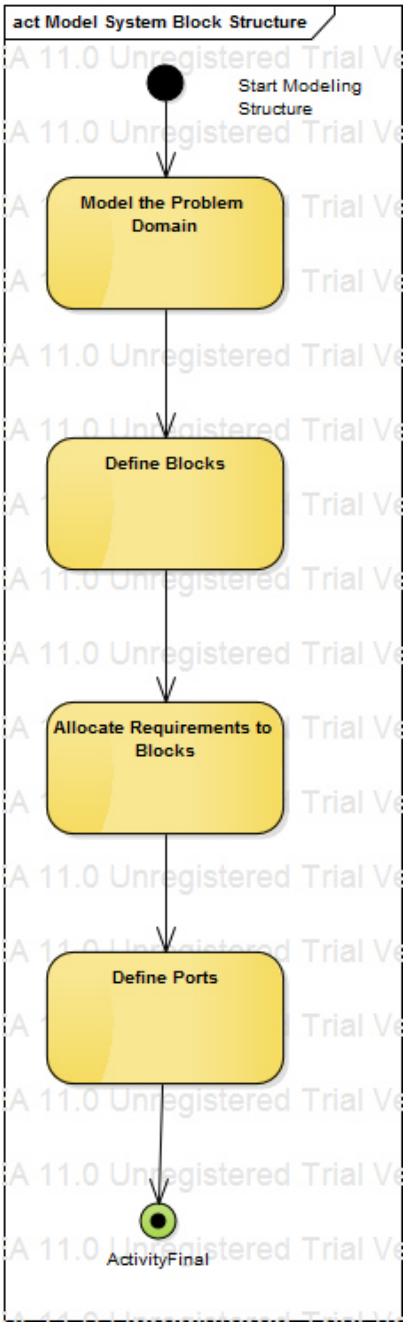

Figure C.4: Structural Model[1]

C.5 Constraints Model

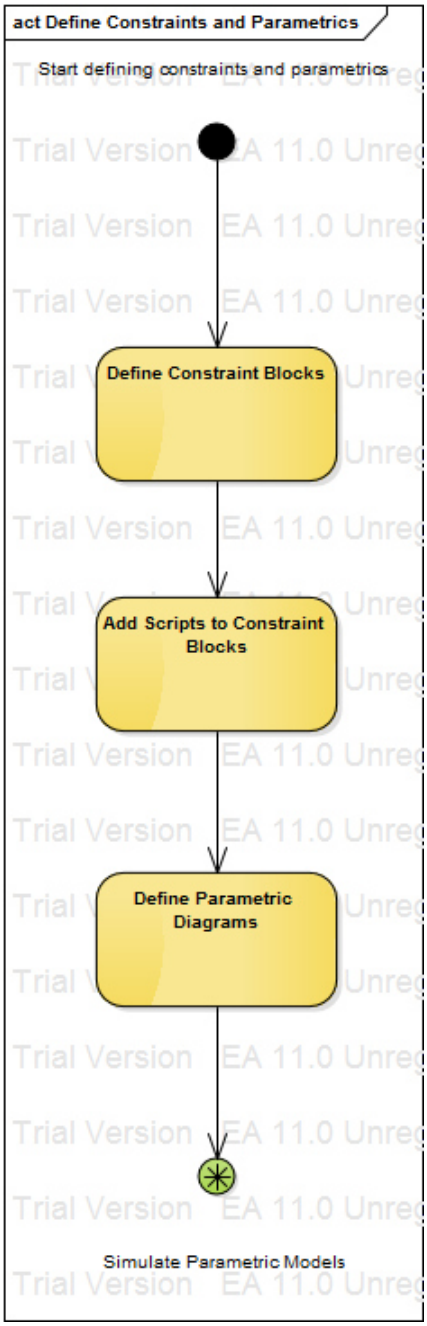

Figure C.5: Constraints Model[1]

Appendix D

Verilog code of the Memory module

```
module Memory #(
  //---- PARAMETER DECLARATION ----------------------------------
  parameter DATA_SIZE = 32 ,
  parameter ADDR_SIZE = 2
  )
  (//---- PORT DECLARATION ----------------------------------------
     input  wire  reset
    ,input  wire  WE
    ,input  wire  clock
    ,input  wire  [ADDR_SIZE-1:0]  address_mem
    ,input  wire  [DATA_SIZE-1:0]  data_in
    ,input  wire  RE
    ,input  wire  [15:0]  LenghtTransfer
    ,input  wire  newXfersize
    ,output  reg  Doorbell
    ,output  reg  [DATA_SIZE-1:0]  Xfersize
    ,output  wire  [DATA_SIZE-1:0]  DAR
    ,output  wire  [DATA_SIZE-1:0]  SAR
    ,output  wire  rstDoorbell
    ,output  reg  [DATA_SIZE-1:0]  data_out
  );

  wire [5:0] w3;
  wire w2;
  wire [ADDR_SIZE-1:0] w5;
  wire [DATA_SIZE-1:0] w1;
  wire w4;
```

```
  MemoryDataPath DataPath (
  .reset(reset)
,.rfsm(w3)
,.WE(WE)
,.WEfsm(w2)
,.addressMEM(w5)
,.XfersizeFSM(w1)
,.Doorbell(Doorbell)
,.data_in(data_in)
,.data_out(data_out)
,.RE(RE)
,.lenghttransfer(LenghtTransfer)
,.Xfersize(Xfersize)
,.DAR(DAR)
,.address_mem(address_mem)
,.SAR(SAR)
,.rstDoorbell(rstDoorbell)
,.doorbellFSM(w4)
,.clock(clock)
  );

  MemoryFSM FSM (
  .reset(reset)
,.rfsm(w3)
,.WE(w2)
,.clock(clock)
,.Doorbell(w4)
,.addressMem(w5)
,.Xfersize(w1)
,.newXfersize(newXfersize)
,.DAR(DAR)
,.SAR(SAR)
,.rstDoorbell(rstDoorbell)
  );

endmodule
```

References

[1] Doug Rosenberg and Sam Mancarella. *Embedded Systems Development using SysML: An Illustrated Example using Enterprise Architect*. Sparx Systems Pty Ltd and ICONIX, 2010. URL: http://www.sparxsystems.com.au/downloads/ebooks/Embedded_Systems_Development_using_SysML.pdf.

[2] OMG. OMG Systems Modeling Language (OMG SysML), Version 1.3, 2012. URL: http://www.omg.org/spec/SysML/1.3/.

[3] David Harel. Statecharts: A visual formalism for complex systems, 1987.

[4] N. Ouerdi, M. Ziane, A. Azizi, and M. Azizi. Modeling embedded systems with sysml. In *Multimedia Computing and Systems (ICMCS), 2012 International Conference on*, pages 905–910, May 2012. doi:10.1109/ICMCS.2012.6320272.

[5] Denis Aulagnier, Ali Koudri, Stéphane Lecomte, Philippe Soulard, Joël Champeau, Jorgiano Vidal, Gilles Perrouin, and Pierre Leray. SoC/SoPC development using MDD and MARTE profile. In Jean-Philippe Babau, Mireille Blay-Fornarino, Joël Champeau, Sèbastien Gèrard, Sylvain Robert, and Antonino Sabetta, editors, *Model Driven Engineering for Distributed Real-time Embedded Systems*. ISTE, 2009. URL: https://hal.inria.fr/inria-00468650.

[6] Mauro Prevostini and Elena Zamsa. Sysml profile for soc design and systemc transformation. *ALaRI, Faculty of Informatics University of Lugano via G. Buffi*, 13(5), 2007.

[7] S. Villa, D. Serna, and J. Aedo. Systemc code generation from uml for wireless sensor networks design.

[8] Fateh Boutekkouk and Okba Fartas. Automatic generation of sysml diagrams from vhdl code.

[9] Emad Ebeid, Franco Fummi, and Davide Quaglia. Hdl code generation from uml/marte sequence diagrams for verification and synthesis. *Design Automation for Embedded Systems*, pages 1–23, 2015. URL: http://dx.doi.org/10.1007/s10617-014-9158-1, doi:10.1007/s10617-014-9158-1.

[10] Marcello Mura, Amrit Panda, and Mauro Prevostini. Executable models and verification from marte and sysml: a comparative study of code generation capabilities. 2008.

[11] Marcello Mura and Marco Paolieri. Sc2 statecharts to systemc: Automatic executable models generation. In *Embedded Systems Specification and Design Languages*, pages 227–239. Springer, 2008.

[12] S. Stancescu, L. Neagoe, R. Marinescu, and E.P. Enoiu. A sysml model for code correction and detection systems. In *MIPRO, 2010 Proceedings of the 33rd International Convention*, pages 189–191, May 2010.

[13] Elvinia Riccobene, Patrizia Scandurra, Alberto Rosti, and Sara Bocchio. A uml 2.0 profile for systemc: toward high-level soc design. In *Proceedings of the 5th ACM international conference on Embedded software*, pages 138–141. ACM, 2005.

[14] Agnes Lanusse, Yann Tanguy, Huascar Espinoza, Chokri Mraidha, Sebastien Gerard, Patrick Tessier, Remi Schnekenburger, Hubert Dubois, and François Terrier. Papyrus uml: an open source toolset for mda. In *Proc. of the Fifth European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA 2009)*, pages 1–4. Citeseer, 2009.

[15] Daniel Knorreck, Ludovic Apvrille, and Pierre de Saqui-Sannes. Tepe: a sysml language for time-constrained property modeling and formal verification. *ACM SIGSOFT Software Engineering Notes*, 36(1):1–8, 2011.

[16] *Modelio user manual 3.3*. http://forge.modelio.org/projects/modelio3-usermanual-english-330/wiki.

[17] J. Bachrach, D. Qumsiyeh, and M. Tobenkin. Hardware scripting in gel. In *Field-Programmable Custom Computing Machines, 2008. FCCM '08. 16th International Symposium on*, pages 13–22, April 2008. doi:10.1109/FCCM.2008.58.

[18] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. Chisel: constructing hardware in a scala embedded language. In Patrick Groeneveld, Donatella Sciuto, and Soha Hassoun, editors, *DAC*, pages 1216–1225. ACM, 2012. URL: http://dblp.uni-trier.de/db/conf/dac/dac2012.html#BachrachVRLWAWA12.

[19] Vassilios Gerousis, Johny Srouji, Karen Pieper, David Smith, Stefen Boyd, Neil Korpusik, Faisal Haque, Steve Meier, Arif Samad, Swapnajit Mittra, Ghassan Khoory, Stuart Sutherland, and Brad Pierce. *SystemVerilog 3.1a Language Reference Manual Accellera's Extentions to Verilog*. Napa, CA, Napa, CA, 2004.

[20] Philippe Coussy and Adam Morawiec. *High-level synthesis*. Springer, 2010.

[21] Hdl coder. http://www.mathworks.com/products/datasheets/pdf/hdl-coder.pdf. Accessed: 2015-03-04.

[22] Matheus Vogel Pinto. Implementação do protocolo can utilizando simulink para geração automática de vhdl. 2011.

[23] Daniele Bagni and Duncan Mackay. Floating-point pid controller design with vivado hls and system generator for dsp. *Xilinx Application Note XAPP1163*, 2013.

[24] Generate from behavioral models. http://www.sparxsystems.com/enterprise_architect_user_guide/9.2/software_engineering/code_generation_from_behaviora.html. Accessed: 2015-03-04.

[25] Màrius Montón, Oriol Font, Jaime Joven, Pere Garcia, Lluís Terés, and Jordi Carrabina. Xml specification and tools for automatic soc generation. In *XIX Conference on Design of Circuits and Integrated Systems*. Citeseer, 2004.

[26] Rohit Kulkarni. Automated rtl generator. 2013.